Heim Verlag

ATARI® ST

J. Teepe

68000 ASSEMBLER

Einführung in die ASSEMBLER-PROGRAMMIERUNG am ATARI ST

Heim Verlag

Auf 3 ½"-Diskette enthalten:
Interaktive Assembler-Entwicklungssoftware



90.1340

68000 Assembler
J. Teepe
– 1. Auflage – Darmstadt: Heim, 1989
ISBN 3-923250-77-O

11
TWS
140

2. Auflage 2010 in Eigenregie
Für Details siehe Seite 399

© Copyright 1989
beim Heim -Verlag · Organisation + Datentechnik
Heidelberger Landstr. 194 · 6100 Darmstadt
Telefon 0 61 51 - 5 60 57



Alle Rechte vorbehalten. Kein Teil dieses Buches darf ohne schriftliche Genehmigung des Heim -Verlages in irgendeiner Form reproduziert oder in eine von Maschinen, insbesondere auch von Datenverarbeitungsmaschinen, verwendete Sprache oder Aufzeichnungs bzw. Wiedergabeart übertragen oder übersetzt werden.

Die Wiedergabe von Warenbezeichnungen, Handelsnamen oder sonstigen Kennzeichen in dem Buch berechtigt nicht zu der Annahme, daß diese von jedermann frei benutzt werden dürfen. Es kann sich auch dann um eingetragene Warenzeichen oder sonstige gesetzlich geschützte Kennzeichen handeln, wenn sie nicht als solche besonders gekennzeichnet sind.

Druck: Druckerei der Heim OHG, 6100 Darmstadt.

Assembler-Programmierung / Atari ST
Atari ST / Assembler-Programmierung
31. 08. 90

I N H A L T S V E R Z E I C H N I S

Inhaltsverzeichnis

ISBN der 1. Auflage 3-923250-77-0

Kapitel 1
Einführung

Dieses Buch vermittelt dem Atari-Besitzer eine Einführung in die Möglichkeiten der Assemblerprogrammierung. Es wird vorausgesetzt, daß er wenigstens eine höhere Computersprache sowie die Bedienung des Betriebssystems beherrscht und mit binären und hexadezimalen Zahlen rechnen kann. Ein Verzeichnis der Fachbegriffe ist in Anhang A aufgenommen.

1.1 ÜBERSICHT DER COMPUTERSPRACHEN

Wenn ich einem Menschen etwas mitteilen möchte, brauche ich dazu eine Sprache. Diese Sprache soll sowohl von mir als auch von meinem Gesprächspartner beherrscht werden.

Wenn ich einem Computer mitteilen möchte, was er für mich auszuführen hat, brauche ich dazu eine Computersprache (so eine Mitteilung bezeichnet man als Programm). Diese Sprache soll sowohl von mir als auch von dem Computer "verstanden" werden. In dieser Sprache kommunizieren also Mensch und Maschine.

Die Programme in den meisten Computersprachen sehen auf den ersten Blick in etwa so aus, wie eine Zwischenform von Englisch und Algebra.

Es gibt - als grobe Einteilung - drei Stufen der Computersprachen:

1. Maschinencode - Assembler
2. Höhere Sprachen
3. Künstliche Intelligenz.

1. Einführung

1.1.1 MASCHINENCODE - ASSEMBLER

Maschinencode ist - wenn man so will - die "natürliche" Sprache des Prozessors.

Mit dem Defehl 0000 0110 0111 1000 0001 0100 0011 0100 0010 0011 0100 0101 wird $1234 zum Speicherplatz $2345 addiert, und mit dem Befehl 0100 1110 1111 1000 0100 0101 0110 0111 wird zum Speicherplatz $4567 gesprungen.

Der Maschinencode ist spezifisch für den Prozessor. Ein Programm in Maschinencode für den 68000 läuft nicht auf dem 80386.

Manche Prozessoren haben einen "kräftigeren" Befehlssatz als andere: Bei dem Prozessor mit dem kräftigeren Befehlssatz kann man mit einem Befehl eine Aufgabe erledigen, für den man mit einem Prozessor mit einem weniger kräftigem Befehlssatz zwei oder mehr Befehle braucht. Der Befehlssatz des 68000 ist kräftiger als der des 8080, aber weniger kräftig als der des 68030.
Der ASSEMBLER ist ein Hilfsprogramm, das jeden Befehl in Assemblersprache in genau einen Befehl in Maschinencode umwandelt.

Wir können also Befehle eingeben wie ADDI.W INCREMENT, BUFFER und JMP CONTINUE. Der Assembler erstellt daraus den richtigen Maschinencode. Der Assembler verbindet das Commando ADD mit 0000 0110 0111 1000, der Wert INCREMENT mit 0001 0100 0011 0100 und die Adresse Buffer mit 0010 0011 0100 0101. Er verbindet den Befehl JMP mit 0100 1110 1111 1000 und der Label CONTINUE mit 0100 0101 0110 0111.

Der Assembler erlaubt uns, mit Kürzeln für die Befehle und mit symbolischen Namen für die Variablen und den Ansprungpunkten zu arbeiten. Er nimmt uns auch die meisten Bittüfteleien ab.

Der Assembler ist spezifisch für den Prozessor. Die Kenntnisse, die Sie sich mit diesem Buch aneignen, sind also nur für den Prozessor 68000 zutreffend (Die meiste Information trifft übrigens auch auf die anderen Mitglieder der "68000-Familie" zu: der 68008, 68010, 68012, 68020, 68030).

1.1.2 HÖHERE SPRACHEN

Höhere Sprachen wie Algol, APL, Basic, C, Cobol, Modula, Pascal und PL1 sind näher zum Menschen und weiter von der Maschine entfernt als Assembler. Ein Befehl in einer höheren Sprache wird in vielen Befehlen auf Maschinencode-Ebene zergliederd. Diese Umsetzung wird gemacht durch ein Hilfsprogramm, das als COMPILER (= Zusammensetzer) bezeichnet wird.

Die höheren Sprachen haben alle ihr eigenes Anwendungsgebiet. Aufgaben im buchhalterischen Bereich löst man z.B. besser mit Cobol, Aufgaben im technisch-wissenschaftlichen Bereich dagegen werden besser in Fortran gelöst. Algol gilt wohl als veraltet, weil die Aufgaben heute besser durch modernere Sprachen gelöst werden können.

Ein in einer höheren Sprache geschriebenes Programm gilt als Maschinen unabhängig: Das bedeutet, daß eine Aufgabe, die Sie auf dem IBM PC in Pascal gelöst haben IM PRINZIP auch auf dem Atari läuft.

Ich habe das IM PRINZIP groß geschrieben, weil in der Praxis die Compiler auf den verschiedenen Maschinen leider nicht voll identisch sind. Die Anpassung eines Programmes, das auf Maschine A ausgetestet wurde, so daß es auf Maschine B läuft, dauert typisch zwischen 2 und 10 % der ursprünglichen Entwicklungszeit des Programms auf Maschine A.

1.1.3 KÜNSTLICHE INTELIGENZ

Die "Künstliche Intelligenz" ist eine "Sprach"-Gruppe, die vom Anwender mitgestaltet wird. Sie liegt noch näher zum Menschen und weiter von der Maschine entfernt, als die höheren Sprachen. Ein wesentlicher Unterschied zu den höheren Sprachen ist, daß man dem Computer mitteilt, welches Ergebnis verlangt wird, anstelle ihm mitzuteilen, wie er zu diesem Ergebnis kommen soll.

1.2 WARUM ASSEMBLER?

Was ist der Grund, warum wir heute noch Programme in Assembler erstellen? Ist Assembler nicht etwas dermaßen antiquiertes, daß wir die Kenntnisse nur noch aus historischen oder nostalgischen Gründen aufheben möchten?

Ganz im Gegenteil! Es gibt gute Gründen, auch dann in Assembler zu programmieren, wenn viele Hochsprachen zur Verfügung stehen. Die Benutzung des Assemblers ergänzt damit die Möglichkeiten, die die höheren Sprachen uns bieten.

1.2.1 COMPILERBAU

Der erste Compiler (er wandelt ein Programm einer höheren Sprache in ein Programm in Maschinensprache um) war in Assembler bzw. Maschinencode geschrieben, weil es z.Zt. der Entwicklung des Compilers noch keinen Compiler gab (das Huhn- und Ei-Effekt).

Man kann aber mit einem provisorischen Assembler einen PLM-Compiler schreiben, und dann anschließend den schicken Macro-Assembler in PLM programmieren (wie Intel es tat). Man kann natürlich einen Basic-Interpreter in C schreiben.

Hiermit wird die Programmierung in Assembler aber nicht überflüssig. Jeder Compiler braucht irgendeinen "Sockel", der in Assembler (bzw. Maschinencode) erstellt wurde. Die Qualität dieses Assembler-Sockels bestimmt die Leistungsfähigkeit des Compilers entscheidend mit.

1.2.2 GESCHWINDIGKEIT

Wenn Sie ein Programm in Assembler erstellen, ist die Ablaufzeit des Programms meistens um ein vielfaches kürzer, als wenn Sie das Programm in einer höheren Sprache schreiben. Als relativ schnell gelten Programme von einem C-Compiler, als ausgesprochen langsam dagegen Programme von einem Basic Interpreter.

Für manche Anwendungen, wie z.B. Interruptbehandlungen innerhalb des Betriebssystems ist eine zu lange Rechenzeit direkt fatal, weil dann Interrupts verloren gehen.

1.2.3 PROGRAMMGRÖSSE

Ein Programm in Assembler ist meistens erheblich kürzer als ein Programm, das in in einer höheren Sprache erstellt wurde.

Extremes Beispiel: Ein Programm das nur den Text

HALLO, WIE GEHT'S?

auf den Bildschirm bringen soll, braucht in Assembler etwa 30 Byte, in C etwa 6 kByte und in Pascal ca. 12 kByte.

1.2.4 EIN- UND AUSGABE

Nicht alle Ein- und Ausgabemöglichkeiten, die die Hardware erlaubt, werden von allen höheren Sprachen unterstützt.

Beispiele:

- Anwahl der Farbe auf Ihrem Bildschirm.
- Abfragen und Setzen der Systemzeit.
- Umschalten der Druck-Ausgaben eines Programms von einem Printerausgang auf den anderen.
- Einschalten Ihrer Kaffeemaschine, die Sie gerade über eine selbstgestrickte Platine an Ihren Computer angeschlossen haben.

Wenn Sie innerhalb Ihres Programms solche Eigenschaften brauchen, müssen Sie aber nicht auf die Anwendung von höheren Sprachen verzichten. Sie können die benötigten Routinen zuerst in Assembler schreiben und austesten. Danach erstellen Sie mit

Hilfe des Assemblers ein Maschinencode-Modul, daß Sie von der Hochsprache aus aufrufen.

Programmteile, die direkt auf das Betriebssystem oder auf die Hardware des Rechners zugreifen, statt über Routinen der Hochsprache, sind betriebssystem- bzw. maschinen-abhängig.

Programme, die direkt auf das Betriebssystem des Atari zugreifen, laufen nicht auf dem IBM PC.

Programme, die direkt auf die ICs des Atari XXX zugreifen, laufen nicht unbedingt auf einem Atari XXY.

1.3 TEST-MÖGLICHKEITEN

Ein unverzichtbares Werkzeug im Werkzeugkasten des Assembler-Programmierers ist der Debugger. Mit diesem Werkzeug hat er ein sehr kräftiges Mittel in der Hand, um in ALLEN Programmen herumzustochern, egal, wer diese Programme geschrieben hat und in welcher Computersprache. Weitere Details in Kap. 1.3.3

1.3.1 DER DEBUGGER

Mit dem Debugger ("Entwanzer") kann er die Fehler (Bugs = "Wanzen") in seinem Programm lokalisieren und beheben.

Der Programmierer kann sein Programm starten und es zunächst an einem Breakpoint kurz vor der verdächtigten Stelle des Programms anhalten lassen.

Danach läßt er das Programm schrittweise weiter laufen: bei jedem Schritt werden die Registerinhalte auf dem Bildschirm angezeigt.

Danach vergleicht der Programmierer den Programmablauf mit seinem Programmlisting. Er sieht nach, weshalb das Programm etwas anderes tut, als er es sich beim Programmieren vorgestellt hatte.

Die Ausbesserungen des Programms sollen natürlich im Programmlisting dokumentiert werden. Mit dem Debugger können die Fehler aber auch provisorisch direkt im Maschinencode behoben werden. Man kann dann weiter testen, ohne den Debugger verlassen zu müssen.

1.3.2 DER DISASSEMBLER DES DEBUGGERS

Der Debugger hat auch eine "Disassemble-Funktion": Er macht quasi das entgegengesetzte von dem, was der Assembler tut. Der Disassembler erstellt also aus Maschinencode eine Art von Assembler-Listing.

Dieses Listing ist aber nicht perfekt, denn der Disassembler kann nicht "wissen", welche Teile des Objectcode Programme und welche Daten sind. Auch die Daten werden als Programm interpretiert, was zu unsinnigen Ergebnissen führt.

BEISPIEL:
Wenn Ihr Disassembler Ihnen z.B:

```
1000   ORI.B  #0,  D0
1002   ORI.B  #0,  D0
1004   ORI.B  #0,  D0
1006   ORI.B  #0,  D0
1008   ORI.B  #0,  D0
100A   ORI.B  #0,  D0     anzeigt,
```

brauchen Sie nicht unbedingt an einen superschlauen Trick eines Ihnen unbekannten Programmierers zu denken. Der Disassembler ist vielmehr in einen Zahlenblock hineingelaufen, der aus lauter Nullen besteht.

In einem Disassembler-Listing sind außerdem die symbolischen Namen, die der Programmierer benutzt hat, nicht mehr vorhanden, und die Kommentare fehlen.

Für den Assemblerprogrammierer, der überprüfen möchte, ob er an der richtigen Stelle seines Programms ist, reicht das Disassembler-Listing allemal.

Ohne Debugger ist es kaum möglich, ein Assembler-Programm auszutesten.

> Die Möglichkeiten des Debuggers sind aber keineswegs zu der Fehlersuche innerhalb selbst erstellter Assemblerprogramme beschränkt.

1.3.3 WEITERE MÖGLICHKEITEN DES DEBUGGERS

Da Assembler die "natürliche" Sprache des Prozessors ist, können Sie sich im Prinzip JEDES Programm mit. dem Disassembler anschauen, ob es nun Teile des Betriebssystems sind, Kompilate oder Programme, von denen nur der ablauffähige Code vorliegt.

Sie können einen Eindruck der Funktionsweise Ihres Compilers erhalten, indem Sie ein Programm schreiben wie z.B.

```
for n := 1 to 3 do
     writeln('hallo');
```

und sich das Ergebnis mit dem Debugger anschauen. Sie werden die Speicheradresse von n finden, und Sie werden sehen, daß die Adresse von 'hallo' auf den Stack gepushed wird, bevor die Funktion writeln aufgerufen wird (die Begriffe "Stack" und "push" werden im Kapitel 6 dieses Buches erläutert).

Sie brauchen den Debugger auch, wenn Programme, die Sie in einer Hochsprache geschrieben haben, nicht laufen und Sie mit den Testmethoden innerhalb der Hochsprache nicht weiter kommen.
Dieser Einsatz des Debuggers stellt also eine Ergänzung der Test-Möglichkeiten des Compiler-Pakets dar. Sie ist leider manchmal notwendig, weil ja nicht alle Compiler immer völlig fehlerfrei arbeiten.

Häufigster Einsatz ist dort, wo Sie z.B. Inkompatibilitäten mit dem Betriebssystem oder mit anderen Programmen vermuten.

Sie können sich mit dem Debugger auch den Maschinencode eines Programms, das Sie nicht erstellt haben, anschauen (vielleicht das Schachprogramm, das Sie bei Ihrem Freund kopiert haben).

Erfahrene Assembler-Programmierer können in so einem Programm notfalls Änderungen durchführen, ohne daß sie über ein Listing des Programms verfügen.

Beispiele:

- o Die Bildschirmtexte innerhalb eines Programms "eindeutschen".
- o Das Programm erweitern (einen Sprung ab der entsprechenden Stelle innerhalb des Programms zu Ihrer Erweiterung und einen Sprung vom Ende der Erweiterung zu der nächsten Position innerhalb des Programms.)
- o Ein geschütztes Programm "knacken". Man knackt ein Programm, wenn man den entsprechenden Schutzmechanismus innerhalb des Programms findet und ihn außer Betrieb stellt.

Solche Änderungen im Object-code sind um Größenordnungen arbeitsintensiver, als wenn Sie das Programm-Listing des entsprechenden Programms vorliegen haben. Wenn die Änderungenswünsche aber nicht allzu groß sind, ist eine Bearbeitung eines bestehenden Programms mit dem Debugger trotzdem einfacher, als wenn Sie das Programm nochmal neu schreiben müßten...

(Die Entscheidung darüber, ob Änderungen an Programmen, die Sie nicht geschrieben haben, juristisch erlaubt sind, liegt aber in der Verantwortung des Programmierers. Wir zeigen Ihnen nur die Werkzeuge.)

1.4 DIE PROGRAMM-ERSTELLUNG

Wie geht nun eine praktische Programm-Erstellung in Assembler vonstatten? Schon etwas komplexer als das erste Mini-Programm in Basic, das mitten in der Ausführung unerwartet stehen bleibt mit:

"SYNTAX ERROR IN 80".

Normalerweise wird für die professionelle Programm-Erstellung ein Entwicklungspaket mit Editor, Assembler, Linker und Debugger benutzt.

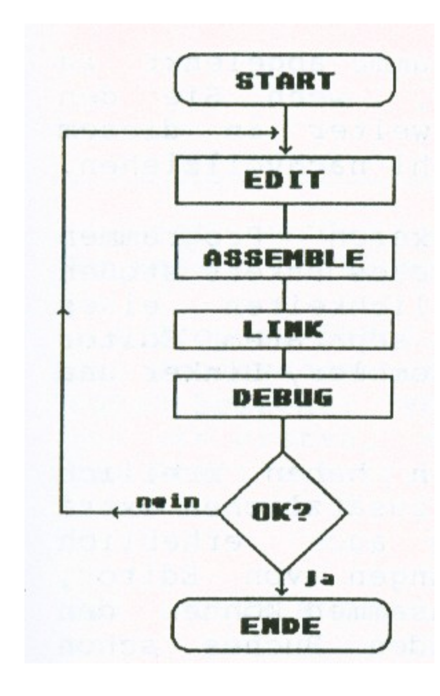
1.4.1 EDITOR, ASSEMBLER, LINKER, DEBUGGER

Sie brauchen einen EDITOR, um Ihr Programm einzutippen. Danach bearbeitet der ASSEMBLER Ihr Programm und meldet Ihnen ggf. formale Fehler innerhalb des Programms. Wenn das so ist, dann müssen Sie mit dem Editor den oder die Fehler verbessern und den Assembler nochmal aufrufen.

Wenn keine formale Fehler auftreten, erstellt der Assembler einen Object-Code.

Der LINKER ("Verbinder") fügt diesen Objectcode zusammen mit anderen - vorher erstellten - Object-Code und schreibt daraus das komplette, ablauffähige Programm. Auch wenn Sie nur einen Objectcode haben, brauchen Sie den Linker, um aus dem Objectcode den Programmcode zu erstellen.

Wenn auch der Linker seine Arbeit ohne Fehlermeldungen getan hat, fängt die Arbeit erst richtig an. Sie haben dann ein Programm vorliegen, das keine formale Fehler enthält. Das bedeutet aber noch lange nicht, daß das Programm dann fehlerfrei funktioniert.



Typischer logischer Ablauf eines Programmiervorgangs

Um das auszutesten, brauchen Sie den DEBUGGER (Siehe Kap. 1.3). Mit dem Debugger lassen Sie Teile des Programms ablaufen und überprüfen, ob die Teile wirklich so arbeiten, wie Sie es erwarten.

Verbesserungen, die Sie im Programm durchführen, sind mit dem Editor anzubringen, so daß danach wieder Assembler, Linker und Debugger aufgerufen werden müssen. Es empfiehlt sich daher, nicht mehr Programm als 1 bis 2 Listingseiten auf einmal zu schreiben und diesen Teil komplett austesten, bevor Sie weitere Programmteile erstellen.

1.4.2 DIE MITGELIEFERTE PROGRAMM-DISKETTE

Um unseren Leser ein einfaches "hereinschnuppern" zu erleichtern, haben wir diesem Buch eine Diskette beigelegt, die ein Mini-Entwicklungspaket beinhaltet. In diesem Paket befinden sich die Funktionen eines Editors, eines Assemblers, eines Linkers und eines Debuggers. Alle diese Funktionen befinden sich gleichzeitig im Hauptspeicher Ihres Ataris, zusammen mit dem zu erstellenden Programmtext und dem Code des zu entwickelnden Programms.

Diese Funktionen sind alle einfach gehalten, so daß Sie sich mit den Möglichkeiten des Assemblerprogrammierens vertraut machen und schnell Programme schreiben können, ohne dabei durch Formalitäten bei

der Bedienung komplexer HiIfsprogramme abgelenkt zu werden. Insbesondere können Sie, wenn Sie den Befehlssatz des 68000 Prozessors weiter in diesem Buch durchlesen, schnell jeden Befehl nachvollziehen.

Bei der Erstellung von komplexeren Programmen werden Sie aber auf die Grenzen dieses Pakets stoßen und nach den zusätzlichen Möglichkeiten eines größeren Entwicklungspakets mit separatem Editor (falls Sie noch keinen haben), Assembler, Linker und Debugger verlangen.

Diese zusätzlichen Möglichkeiten haben freilich auch Ihren Preis: Durch die vielen zusätzlichen extra Möglichkeiten ist die Bedienung auch erheblich komplexer. Die Bedienungsanleitungen von Editor, Assembler, Linker und Debugger zusammen können den Umfang des Ihnen jetzt vorliegenden Buches schon übertreffen.

Sobald Sie die Grenzen dieses Einsteigerpakets spüren, haben wir unser Ziel allerdings schon erreicht, denn dieses Buch möchte Ihnen einen Einstieg in die Assemblerprogrammierung des 68000 geben.

Die Bedienungsanleitung der Programmdiskette ist in Anhang C aufgenommen.

Kapitel 2

Architektur

Jetzt werden wir uns ansehen, welche Teile der Prozessor 68000 enthält.

A Propos, 68000. Es gibt die 68000-Familie, die aus den Prozessoren 68000, 68008, 68010, 68012, 68020 und 68030 besteht. Von diesen Prozessoren besprechen wir nur den ersten, den 68000, weil dieser Prozessor in dem Atari steckt.

2.1 USER MODE - SUPERVISOR MODE

Der Prozessor hat zwei Betriebsarten, den User Mode und den Supervisor Mode.

In Anwendungsprogrammen befindet sich der Prozessor normalerweise im User Mode. Im User Mode sind einige Kommandos des Befehlssatzes ausgeklammert: Diese laufen nur im Supervisor Mode. Hierdurch wird die Gefahr, daß z.B. bei einem Systemabsturz ungewollt Files auf der Platte gelöscht werden (was uns bei einem anderen Prozessor mal passierte), erheblich verringert.

Ein Anwendungsprogramm kann den Prozessor aber durch Aufruf eines entsprechenden Kommandos, z.B. TRAP oder ILLEGAL in den Supervisor Mode umschalten: der Programmierer sollte das nur dann tun, wenn er dringende Gründe dazu hat. Normalerweise wird der Supervisor Mode nur innerhalb des Betriebssystem angewählt.

Weitere Details über die Umschaltung in den Supervisor Mode bei Exceptions, Kap. 6.

Die folgenden Befehle (s.g. privilegierte Befehle) laufen nur in Supervisor Mode ab:

- o Zugriffe zum Sytem Byte des Status Registers
- o RESET Alle externen Geräte werden zurück gesetzt.
- o RTE Return von einem "Exception Condition"
- o STOP Hält das Programm an, bis ein externes Ereignis erfolgt.

Eine komplette Liste der privilegierten Befehle ist in Kap. 4.2.8 enthalten.

2.2 DIE REGISTER

Hier sehen wir die Register des 68000 Prozessors.

Der Supervisor Stack Pointer sowie das System Byte des Status Registers sind nur im Supervisor Mode zugänglich. Diese beiden Register sind in Bild 2.1 gerastert unterlegt.

Die Bits werden von rechts nach links nummeriert, wobei Bit 0 die Wertigkeit $2^0=1$, und Bit 31 die Wertigkeit 2^{31} = 2 147 483 648 hat.

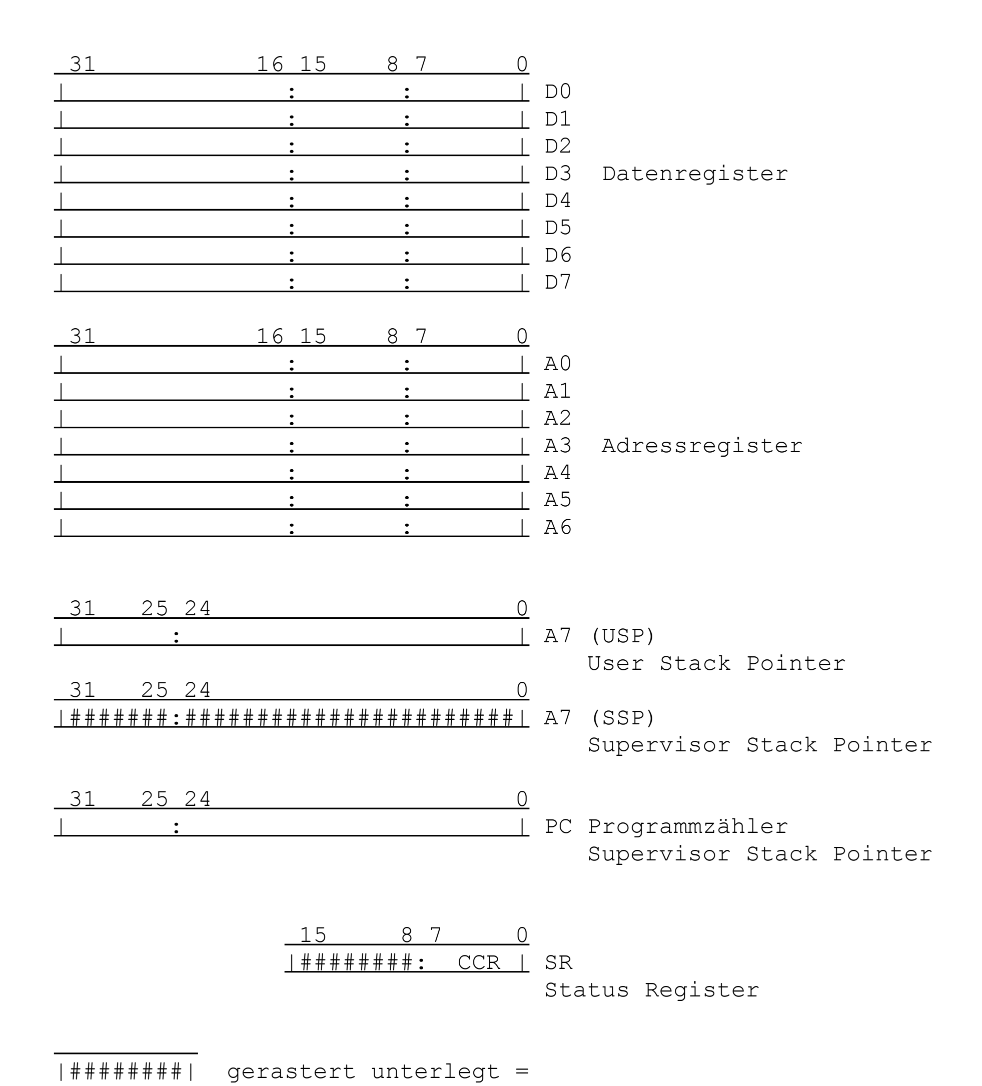


Bild 2.1 Die Register des Prozessors 68000

2.2.1 DIE DATENREGISTER

In den Datenregistern D0..D7 können wir Daten von 8, 16 oder 32 Bits Datenlänge abspeichern. Wenn Daten von 8 oder 16 Bits Datenlänge im Register abgespeichert werden, ändern sich nur die 8 bzw. 16 niederwertigen Bits, die nicht betroffenen hochwertigen Bits behalten ihren bisherigen Wert.
Es gibt 8 Datenregister von 32 Bits. Sie werden von dem Befehlssatz her völlig gleichwertig behandelt.

2.2.2 DIE ADDRESSREGISTER

In den Adressregistern A0..A7 können Sie Adressen (oder auch Daten) von 32 Bits abspeichern. Wenn das Register gelesen wird, werden hiervon - je nach Operator - 16 oder 32 Bits ausgewertet. Bei jedem Schreibvorgang werden aber alle 32 Bits des Adressregisters beschrieben.

Von den Adressen des Prozessors 68000 sind die hochwertigen 8 Bits hardwaremäßig nicht implementiert (siehe Kap. 3.1). Das bedeutet, daß der Inhalt von Bit 0..7 der Datenregister für die Adressierung der Daten unwichtig ist.

Wir empfehlen aber aus Gründen von Transparenz und Kompatibilität, diesen 8 Bits den Wert 0 mitzugeben, wenn Sie über die Adressregister Daten adressieren möchten.

Es gibt sieben Adressregister von 32 Bits, A0 bis A6.

Es gibt darüber hinaus z w e i v e r s c h i e d e n e Adressregister A7, die als Stack Pointer benutzt werden können, je nachdem, ob der Prozessor sich im User Mode oder im Supervisor Mode befindet.

Diese beiden Register A7 werden daher als U s e r S t a c k P o i n t e r und als S u p e r v i s o r S t a c k P o i n t e r bezeichnet. Mehr über den Betrieb des Stackpointers in Kap. 6.

2.2.3 DER PROGRAMMZÄHLER

Der Programmzähler ("Program Counter") PC enthält. die Adresse des nächsten Befehls. Eine explizite Änderung erfolgt z.B. durch bedingte und unbedingte Sprunganweisungen und durch Aufrufe von Unterprogrammen.

Bei dem Program Counter sind - wie bei den Adressregistern - die acht hochwertigen Bits unerheblich.

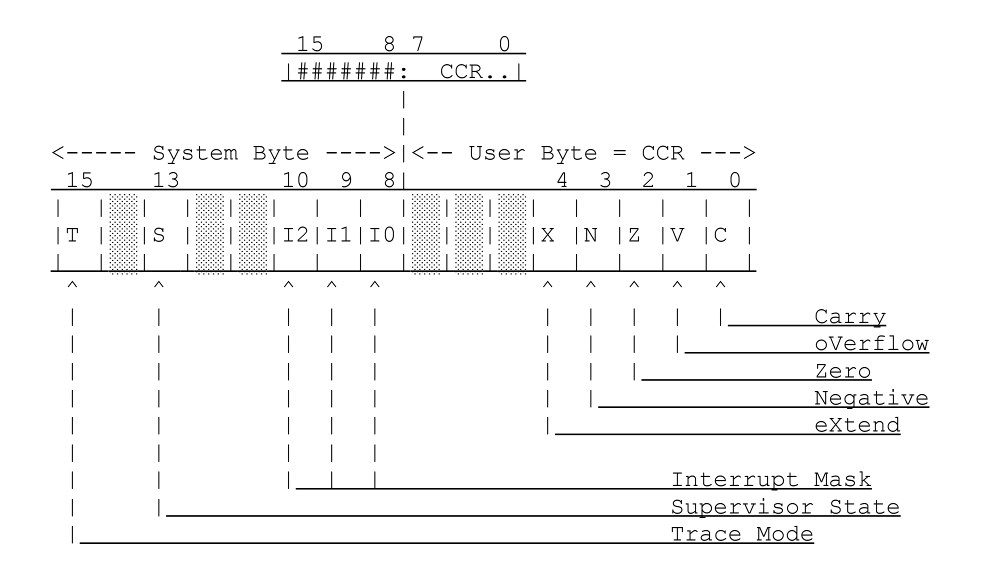
2.2.4 D A S S T A T U S R E G I S T E R

Das Statusregister SR besteht aus zwei Bytes:

- o Das Condition Code Register (CCR) oder das User Byte. Es ist immer zugänglich.
- o das System Byte, Es ist nur im Supervisor Mode zugänglich.

Viele Operationen - jedoch nicht alle - haben eine Änderung des Status Registers zufolge. Genaue Angaben darüber, welche Operationen eine Änderung des Status Registers bewirken und welche nicht, sind in der Befehlsübersicht in Anhang B enthalten.

Das Statusregister wird durch bedingte Sprünge und bedingte Anrufe eines Unterprogramms etc. ausgewertet.




```
|############| gerastert unterlegt =
|############| Register ist nur in Supervisor Mode erreichbar.
```

Bild 2.2 Das Status Register des Prozessors 68000

2.2.4.1 DAS USER BYTE = CONDITION CODE REGISTER

Das User Byte innerhalb des Status Wortes besteht aus dem Status Bits Carry, Overflow, Zero, Negative und Extend.

Das User Byte wird auch als **Condition Code Register** (CCR) bezeichnet und die einzelnen Bits als **Condition Code.**

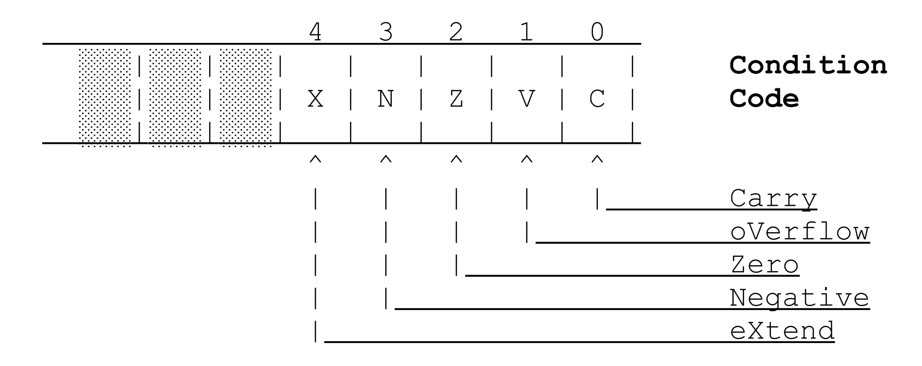


Bild 2.3
Statusregister: Bit-Zuordnung des User Bytes.

Das C-Bit (Carry = Übertrag) führt das höchstwertige Bit einer arithmetischen Operation. Wenn z.B. zwei Zahlen von 10 Bit addiert werden, erhält das C Register den Wert von Bit 11 des Ergebnisses. Das Carry-Bit empfängt auch das hinausgeschobene Bit bei Schiebe-Operationen.

Das V-Bit (oVerflow = Überlauf) wird gesetzt, wenn das Ergebnis einer Operation nicht richtig dargestellt werden kann. Wenn z.B. $7FFF und $03 addiert werden, dann ist $8002 nicht die richtige Darstellung des Ergebnisses. (Denken Sie daran: $8002 in der Zweikomplement-Darstellung entspricht -32765 dezimal). Bei dieser Operation wird das V-Bit gesetzt.

Das Z-Bit (Zero = Null) wird gesetzt, wenn das Ergebnis einer Operation gleich Null ist.

Das N-Bit (Negative) wird gesetzt, wenn das höchstwertige Bit in einer Operation gesetzt wird.

Das X-Bit (eXtended = erweitert) ist eine Kopie des C-bits (siehe oben); Es wird aber nicht immer verändert, wenn das Carry-Bit sich ändert. Das X-Bit wird nur durch solche Operationen, die für Rechnen mit hoher Genauigkeit gedacht sind, beeinflußt.

2.2.4.2 DAS SYSTEM BYTE

Das System Byte innerhalb des Status Wortes besteht aus der Interrupt Maske und aus den Status Bits Supervisor State und Trace Mode.

Während das User Byte (siehe oben) immer zugänglich ist, ist das System Byte nur dann erreichbar, wenn der Prozessor sich im Supervisor Mode befindet.

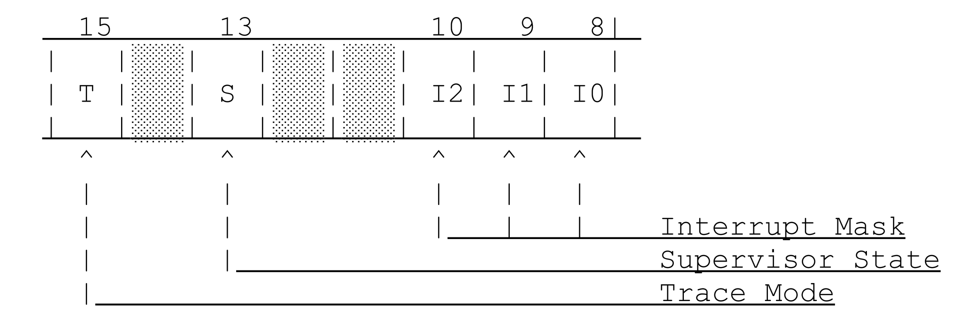


Bild 2.4
Statusregister: Bit-Zuordnung des System Bytes.

Die Bits I_2, I_1 und I_0 bilden eine Interrupt Maske, die bestimmt, welche Hardware Interrupts bedient werden und welche nicht. Mehr über Interrupts bei der Besprechung von Exceptions in Kap. 6.

Das S-Bit (Supervisor-Bit) bestimmt, ob der Prozessor sich in Supervisor Mode (S=1) oder in User Mode (S=0) befindet. Der Unterschied zwischen Supervisor Mode und User Mode wird in Kap. 2.1 besprochen.

Das T-Bit (Trace-Bit) bestimmt, ob der Prozessor nach jeder Instruktion eine Exception macht (T=1) oder nicht (T=0). Durch die Benutzung des Trace-Bit wird es einfach, einen Debugger zu schreiben. Mehr Information über Exceptions in Kap. 6.

Kapitel 3
Datenstrukturen

3.1 DER ADRESSBEREICH

Die 68000-Familie verfügt über einen Adressraum von 32 Bits, so daß ein Adressbereich von 2 Bytes, also 4 294 967 296, zur Verfügung steht. Innerhalb von diesem Speicherplatz sind sämtliche Befehle, sämtliche Daten und alle Eingabe-Ausgabe-Kanäle untergebracht.

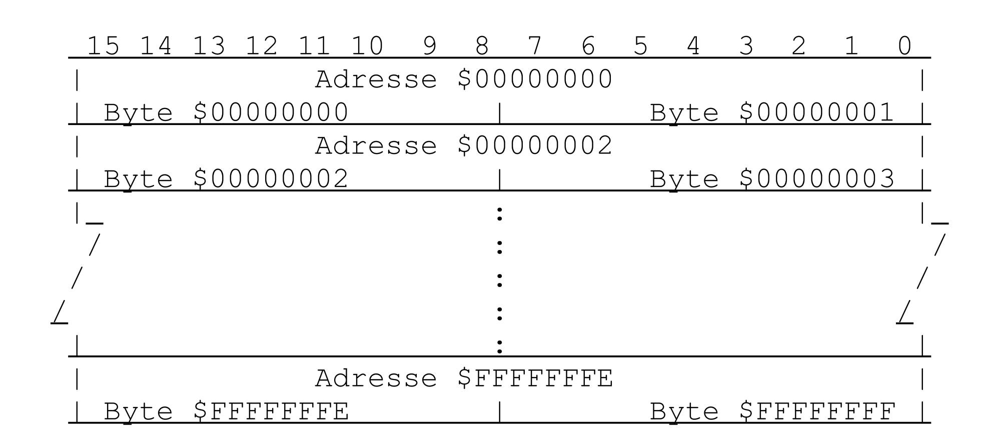


Bild 3.1 Darstellung des Adressierungsbereichs

Da wir von dieser Prozessorfamilie in diesem Buch nur den Prozessor 68000 besprechen werden, sei erwähnt, daß die hochwertigen acht Adressbits bei dem 68000 nicht implementiert sind, wohl aber bei den größeren Brüdern, dem 68020 und dem 68030.

Wer aber auf Kompatibilität seiner Programme mit den größeren Prozessoren achtet, oder wer für klaren Programmierstil ist, möchte doch bitte dafür sorgen, daß die acht hochwertigen Bits alle null sind.

Hier sehen wir, wie der Adressbereich organisiert ist. Zur Verfügung steht ein Adressraum von 32 Bytes, so daß 2 Bytes bzw. 2 Wörter adressiert werden können. Der Adressbereich läuft dann von $00000000 bis 5FFFFFFFE.

Innerhalb von diesem Adressbereich sind sämtliche Daten und Befehle, sowie alle Ein- und Ausgabe-Kanäle untergebracht.

3.2 DATENORGANISATION IM SPEICHER

Wir haben hier oben gesehen, daß der Adressierungsraum durch 32 Bit festgelegt wird.

Um im Speicher unsere Daten aufzufinden, müssen wir genau wissen, wie die Daten dort abgelegt werden.

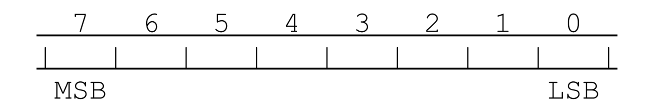


Bild 3.2 Bit-Numerierung eines Bytes

Bei einem Byte werden die Bits von rechts nach links nummeriert, wobei Bit 0 die Wertigkeit 2^0=1 und Bit 7 die Wertigkeit 2^7 =128 hat.

3.3 ARTEN VON DATEN

Welche Art von Daten kann der 68000 nun in seinem Befehlssatz verarbeiten?

Es sind die folgenden:

- o Bytes (8 Bit)
- o Wörter (16 Bit)
- o Langwörter (32 Bit)
- o BCD-Zahlen (4 Bit)
- o Einzelne Bits

Außerdem ist in dem Befehlssatz des 68000 die Verarbeitung von anderen Datentypen wie Speicheradressen, Inhalte von Statuswörter usw. vorgesehen.

Die Adressierung der Bytes, Wörter, Langwörter und BCD-Zahlen wird nachstehend besprochen, ebenso wie ein Beispiel der Abspeichermöglichkeiten von Strings.

Für die Adressierung der einzelne Bits siehe im Anhang B bei den Befehlen BCHG, BCLR, BSET und BTST nach.

3.4 BEZEICHNUNG VON DATENTYPEN IM BEFEHLS-SYNTAX

Bei den meisten Befehlen muß angegeben werden, ob der Operand nun ein Byte, ein Wort oder ein Langwort ist. Wir geben dies mit der Endung ".B", ".W" oder ".L" an.

```
MOVE.B   kopiert ein Byte (8 Bit)
MOVE.W   kopiert ein Wort (16 Bit)
MOVE.L   kopiert ein Langwort (32 Bit)
```

3.5 ADRESSIERUNG VON BYTES, WÖRTERN UND LANGWÖRTERN

3.5.1 DARSTELLUNG IN 8-BIT

Obwohl der 68000 ein 16-bit Prozessor ist, stellen wir ihn - aus didaktischen Gründen - zunächst vor, als wäre er ein 8-Bit-Prozessor.

> (Bedenken Sie dabei, daß Sie, ohne Ihren Computer zu öffnen und auf die Platine zu schauen, niemals feststellen können, ob sich darin nun ein 8-Bit oder ein 16-Bit Datenbus befindet, da das logische Verhalten der beiden Prozessoren-Typen voll identisch ist.)

Die hier gegebene 8-Bit-Vorstellung ist tatsächlich für den 8-Bit-Prozessor der 68000 Familie, den 68008 zutreffend.

Wir benutzen hier die Adresse 1000 als Beispiel für die Lage der Adressen im Speicher. So umgehen wir die Benutzung von irreführenden Wörtern wie "oben" und "unten".

```
        7    6    5    4    3    2    1    0
1000   | Byte 0                              |
1001   | Byte 1                              |
1002   | Byte 2                              |
1003   | Byte 3                              |
1004   | Byte 4                              |
1005   | Byte 5                              |
1006   | Byte 6                              |
1007   | Byte 7                              |
        MSB                                LSB
```

Bild 3.3 Adressierung nachfolgender Bytes (8-Bit)

MSB = most significant bit (hochwertig)
LSB = least significant bit (niederwertig)

```
       7    6    5    4    3    2    1    0
1000  | Byte 0 hochwertiges Byte     Wort   |
      | Byte 1 niederwert. Byte       0     |
1002  | Byte 0 hochwertiges Byte     Wort   |
      | Byte 1 niederwert. Byte       1     |
1004  | Byte 0 hochwertiges Byte     Wort   |
      | Byte 1 niederwert. Byte       2     |
       MSB                                LSB
```

Bild 3.4 Adressierung nachfolgender Wörter (8-Bit)

Wir sehen hier, daß das hochwertige Byte eines Wortes auf die niedrige Adresse abgelegt wird. Wenn ich also das Wort $1234 in die Speicheradresse $1000 abspeichere, dann kommt $12 in $1000, und $34 in $1001.

> Programmierer, die früher mit Intel gearbeitet haben, müssen hier umdenken, denn sie sind gewöhnt, daß $12 (bzw 012H) in $1001 und $34 in $1000 abgespeichert wird.

```
        7    6    5    4    3    2    1    0
      | Byte 0                    hochwertiges   |
1000  | Byte 1      Langwort                Wort |
      | Byte 2         0        niederwertiges   |
      | Byte 3                              Wort |
      | Byte 0                    hochwertiges   |
1004  | Byte 1      Langwort                Wort |
      | Byte 2         1        niederwertiges   |
      | Byte 3                              Wort |
       MSB                                  LSB
```

Bild 3.5 Adressierung nachfolgender Langwörter. (8-Bit)

Auch die Adressierung der Langwörter ist hier konsequent durchgeführt: die hochwertigsten Teile kommen an die niedrigste Adresse.

3.5.2 DARSTELLUNG IN 16-BIT

Wenn wir jetzt berücksichtigen, daß wir tatsächlich einen 16-Bit-Prozessor statt einem 8-Bit-Prozessor haben, erscheinen die nachfolgenden Diagramme.

Die Erscheinung nach außen zum Programmierer ist aber genau die gleiche, als die hier oben in 3.5.1 beschrieben.

Deswegen auch hier:
Achtung Intel-Programmierer!

```
      15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
      |       Byte 0          |       Byte 1          |
1000  | MSB              LSB  |                       |
      |       Byte 2          |       Byte 3          |
1002  | MSB              LSB  |                       |
      |       Byte 4          |       Byte 5          |
1004  | MSB              LSB  |                       |
```

Bild 3.6   Adressierung nachfolgender Bytes (8-Bit)

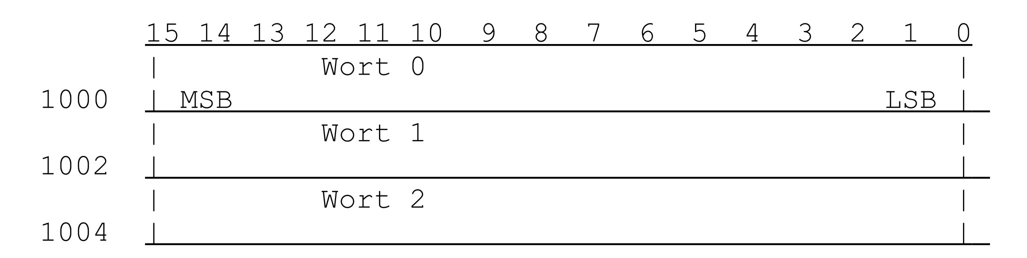


Bild 3.7 Adressierung nachfolgender Wörter (16-Bit)

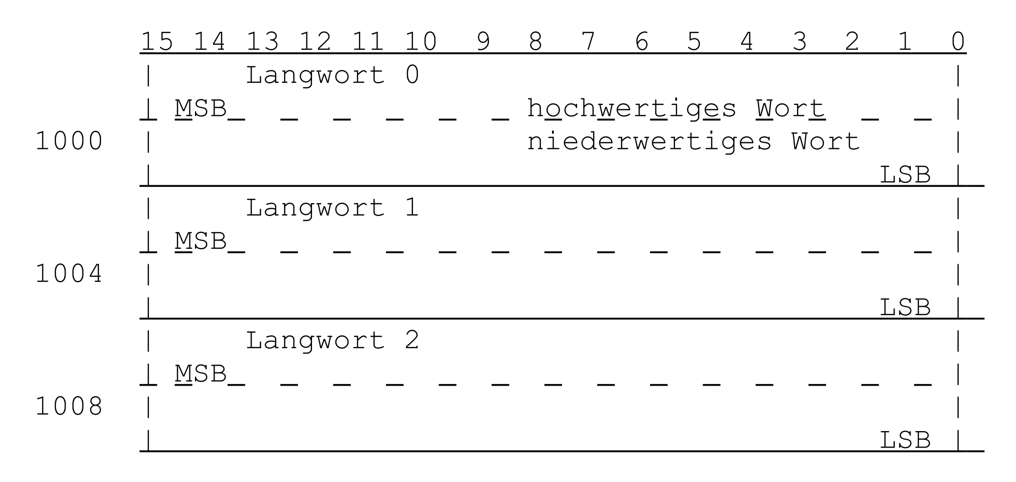


Bild 3.8 Adressierung nachfolgender Langwörter (32-Bit)

3.5.3 VORZEICHEN VON BYTES, WÖRTERN UND LANGWORTERN

Wir haben gesehen, daß binäre Daten verarbeitet werden können:

o als Byte (8 Bit);
o als Wort (16 Bit);
o als Langwort 32 Bit).

Die Angabe, ob der Prozessor nun ein Byte, ein Wort oder ein Langwort verarbeiten soll, finden wir im letzten Buchstaben des Befehls. So bedeutet ".B" in dem Befehl "MOVE.B D2, D4", daß die Operandgröße ein Byte ist.

Es gibt noch eine Unterscheidungsart, wie wir die Bits im Speicher unseres Computers verstehen können, nämlich:

o als Binärzahl ohne Vorzeichen;
o als Binärzahl mit Vorzeichen.

Auch hier ist es der Computerbefehl, der entscheidet, ob die Binärzahl als Binärzahl mit oder ohne Vorzeichen aufgefaßt wird.

Bei den meisten Befehlen ist es in der Abarbeitung egal, ob die Zahl als Binärzahl mit oder ohne Vorzeichen zu verstehen ist. So ist bei ADD und SUB-Befehlen das Ergebnis gleichermaßen richtig.

Bei den Befehlen, wo ein Unterschied gemacht wird, wird bei der Befehlsbeschreibung in Anhang B darauf hingewiesen.

Das Vorzeichenbit ist das hochwertige Bit der Daten. Bei einem Byte ist das also Bit 7, bei einem Wort Bit 15 und bei einem Langwort Bit 31.

Ist das Vorzeichenbit zurückgesetzt (0), dann ist der Zahl positiv oder null. Ist das Vorzeichenbit gesetzt (1), dann ist die Zahl negativ.

Die hier beschriebene Darstellung von Zahlen mit Vorzeichenbit wird auch als **Zweierkomplement** bzw. **2-complement** bezeichnet.

Am Beispiel eines Bytes möchten wir Ihnen den Unterschied zwischen den beiden Interpretationen zeigen:

Binärzahl Bit 76543210	Hex-Wert	aufgefaßt als Binärzahl mit Vorzeichen	aufgefaßt als Binärzahl ohne Vorzeichen
00000011	$03	3	3
00000010	$02	2	2
00000001	$01	1	1
00000000	$00	0	0
11111111	$FF	-1	255
11111110	$FE	-2	254
10000001	$81	-127	129
10000000	$80	-128	128
01111111	$7F	127	127
01111110	$7E	126	126

3. Datenstrukturen

Bei Wörtern und Langwörtern läuft das Verfahren völlig analog. Hier ist die Darstellung für ein Wort:

Binärzahl Bit FEDCBA9876543210	Hex- Wert	Binär- zahl mit Vor- zeichen	Binär- zahl ohne Vor- zeichen
0000000000000011	$0003	3	3
0000000000000010	$0002	2	2
0000000000000001	$0001	1	1
0000000000000000	$0000	0	0
1111111111111111	$FFFF	-1	65535
1111111111111110	$FFFE	-2	65534
1000000000000001	$8001	-32767	32769
1000000000000000	$8000	-32768	32768
0111111111111111	$7FFF	32767	32767
0111111111111110	$7FFE	32766	32766

Eine Zahl wird invertiert (von positiv zu negativ oder von negativ zu positiv), indem man zuerst alle Bits invertiert und danach zum Ergebnis 1 addiert.

Sie brauchen das nicht selber auszutüfteln, der Befehl NEG macht das schnell und problemlos für Sie.

3.5.4 UMWANDLUNG VON BINÄRZAHLEN MIT VORZEICHENBIT

Wenn wir Binärzahlen mit Vorzeichen umwandeln möchten, z.B. von Langwort zu Byte oder von Wort zu Langwort, müssen wir aber aufpassen, daß nichts falsch läuft.

3.5.4.1 UMWANDLUNG VON KLEINEN ZU GROSSEN FORMATEN

Wenn wir Binärzahlen mit Vorzeichen von kleinen zu großen Formaten umwandeln möchten, z.B. von Byte zu Langwort, geht das relativ einfach.

Wir kopieren das Byte rechtsbündig in das Langwort hinein und kopieren zusätzlich das Vorzeichenbit in alle nicht belegten Bitpositionen.

```
BEISPIEL 1:             Vorzeichenbit
                        ▼
                    Bit 76543210
                        00000011 Binärzahl +3 in Byte
                0000000000000011          zum Wort
00000000000000000000000000000011       zum Langwort
```

```
BEISPIEL 2:     Vorzeichenbit
                ▼
           Bit 15.....9876543210
                1111111111111100 Binärzahl -4 in Wort
11111111111111111111111111111100       zum Langwort
```

Sie brauchen das nicht selber auszutüfteln, der Befehl EXT führt das problemlos für Sie durch.

3.5.4.2 UMWANDLUNG VON GROSSEN ZU KLEINEN FORMATEN

Wenn wir Binärzahlen mit Vorzeichen von großen zu kleinen Formaten umwandeln möchten, z.B. von Byte zu Langwort, geht das noch einfacher, indem wir die nicht benötigten Bits an der linken Seite (die hochwertigen Bits) weglassen.

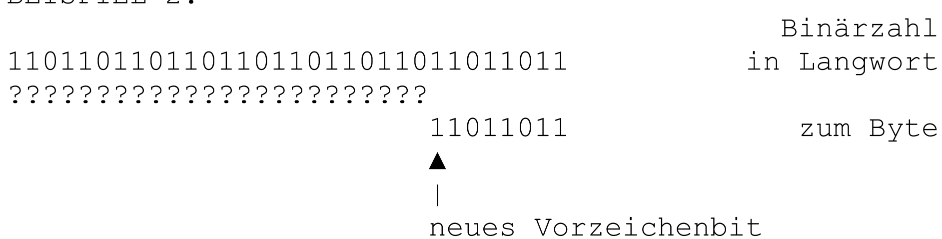
Wir DÜRFEN das aber nur machen, wenn **alle** hochwertigen Bits den **gleichen** Wert haben wie das neue Vorzeichenbit.

BEISPIEL 1:

```
          Bit 15.....9876543210
               1111111111111100 Binärzahl -4 in Wort
                       11111100            zum Byte
                       ▲
                       |
                       neues Vorzeichenbit
```

Diese Umwandlung ist zulässig, weil alle Bits auf den Positionen F..8 den **gleichen** Wert haben wie das neue Vorzeichenbit.

BEISPIEL 2:



Diese Umwandlung ist nicht zulässig, weil die mit "?" gekennzeichneten Bits nicht **alle** den **gleichen** Wert haben wie das neue Vorzeichenbit.

Logisch: denn der Wert im Langwort ist in einem Byte überhaupt nicht darstellbar.

3.6 ADRESSIERUNG VON BCD-ZIFFERN

```
      15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
1000  |  BCD 0    |    BCD 1    |    BCD 2    |    BCD 3   |
      | MSD  LSD  |             |             |            |
1000  |  BCD 4    |    BCD 5    |    BCD 6    |    BCD 7   |
      |           |             |             |            |
```

Bild 3.9 Adressierung nachfolgender BCD-Ziffern (16-Bit)

Und hier sieht man, daß auch bei BCD-Ziffern die hochwertigsten Teile an der niedrigsten Adresse abgespeichert werden. Der 68000 benutzt diese Adressierung bei den Befehlen ABCD, SBCD und NBCD, siehe Anhang B.

Und wie rechnet man nun mit BCD-Ziffern?

Für jede BCD-Ziffer stehen 4 Bit zur Verfügung. Die gültigen Werte innerhalb einer BCD-Ziffer sind \$0..\$9, die den dezimalen Ziffern 0 bis 9 entsprechen. Die Werte \$A..\$F sind nicht zugelassen.

So erhält die dezimale Zahl 987654 gemäß Bild 3.9 die folgenden Werte:

BCD Ziffer	0	1	2	3	4	5	6	7
Inhalt	\$0	\$0	\$9	\$8	\$7	\$6	\$5	\$4
Inhalt binär	0000	0000	1001	1000	0111	0110	0101	0100

Dagegen entspricht die dezimale Zahl 987654 dem Hex-Wert \$F1206, der als Langwort (siehe Bild 3.8), ganz anders abgespeichert wird:

```
0000 0000 0000 1111      ($000F)
0001 0010 0000 0110      ($1206)
```

Die Länge einer BCD-Zahl kann vom Programmierer beliebig festgelegt werden.

Vorteil vom Rechnen mit BCD-Zahlen statt mit binären Zahlen ist, daß bei Multiplikationen Rundungsfehler besser vermieden werden können, je nach verwendetem Algoritmus. Auch können Konvertierungen zu und von Zeichenketten einfacher durchgeführt werden.

Nachteile sind aber vermehrter Speicherbedarf und Rechenzeit.

3.7 ADRESSIERUNG VON STRINGS (ZEICHENKETTEN)

Eine Vorgabe, ob ein String an der niedrigsten oder an der höchsten Adresse anfängt, gibt es für den 6800Ü-Programmierer nicht. Es liegt auch im Ermessen des Programmierers, wie er das Ende des Strings erkennbar macht.

Üblich bei der "C"-Programmierung ist, daß eine Zeichenkette bei der niedrigen Adresse anfängt und bei der höheren Adresse aufhört. Das Ende wird durch ein $0-Zeichen gekennzeichnet.

```
BEISPIEL:
1000  53 6F 20 65 69 6E 20 54   So ein T
1008  61 67 2C 20 73 6F 20 77   ag, so w
1010  75 6E 64 65 72 73 63 68   untersch
1018  6F 65 6E 20 77 69 65 20   oen wie
1020  68 65 75 74 65 00 AF DC   heute...
```

3.8 BEFEHLSSTRUKTUR IM SPEICHER

Jetzt schauen wir uns an, wie ein Befehl konkret im Speicher des Computers abgespeichert wird.

Als Beispiel nehmen wir uns den Befehl MOVE vor.

Auch weiterhin werden wir in diesem Buch häufig MOVE als Beispiel benutzen, weil man daran so schön alle Adressierungsarten zeigen kann.

MOVE kopiert Daten (Byte, Wort oder Langwort) von einer Speicherstelle zur anderen. Die zu kopierende Datenmenge beträgt:

```
bei  MOVE.B  8 Bit    (B = Byte)
bei  MOVE.W  16 Bit   (W = Wort)
bei  MOVE.L  32 Bit   (L = Langwort).
```

Wir haben MOVE mit "kopieren" übersetzt, weil die Quelldaten unverändert im Speicher stehen bleiben.

3.8.1 BEFEHLE OHNE ARGUMENTWORT

Der Befehl
MOVE.L D3,D5 (kopiere Datenregister D3 nach D5)

bildet im Speicher ein Befehlswort von 16 Bit Länge, unabhängig von der zu kopierenden Datenmenge.

```
Adresse  _________________
1000     | MOVE.L D3, D5 |        Befehlswort
```

Im Speicher wird direkt nach diesem Befehlswort das nächste Befehlswort angeordnet, z.B.
ADD.L D2, D5

Das zweite Befehlswort erhält im Speicher eine Adresse, die um 16 Bit = 1 Wort = 2 Bytes höher liegt.

```
Adresse  _________________
1000     | MOVE.L D3, D5 |        Befehlswort
1002     | ADD.L D2, D5  |        nächstes Befehlswort
```

Das Wort, welches dem Befehlswort folgt, wird als nächstes Befehlswort genommen.

3.8.2 BEFEHLE MIT EINEM ARGUMENTWORT

In dem Befehl

MOVE.B #$12,D5 (schreibe die Zahl 12 hex in D5)

wird dem Befehlswort ein zweites Wort angehängt, das das Argument des Befehls trägt.

```
Adresse  ______________________
1000     | MOVE.B #<data>, D5 |   Befehlswort
1002     |     00          12  |   1. Argumentwort
1004     | ADD.L D2, D5        |   nächstes Befehlswort
```

In der niederwertigen Hälfte des Argumentwortes steht in diesem Falle die Zahl $12, die höherwertige Hälfte des Argumentwortes ist mit Nullen gefüllt.

An dem Befehlswort "sieht" der Prozessor, daß das dem Befehlswort unmittelbar folgende Wort als Argument zu nehmen ist.

Von dem Argument liest er nur die niederwertigen 8 Bits, diese werden im Register D5 abgespeichert.

Das dem 1. Argumentwort folgende Wort wird als nächstes Befehlswort genommen.

3.8.3 BEFEHLE MIT ZWEI ARGUMENTWÖRTERN

In dem Befehl

```
       MOVE.L #$12345678, D5
           (schreibe die Zahl 12345678 in D5)
```

werden dem Befehlswort zwei Wörter angehängt, Sie tragen das Argument des Befehls.

```
Adresse  ______________________
1000     | MOVE.L #<data>, D5 |   Befehlswort
1002     |    12          34  |   1. Argumentwort
1004     |    56          78  |   1. Argumentwort
1006     | ADD.L D2, D5       |   nächstes Befehlswort
```

3.8.4 QUELLE UND ZIEL BESITZEN ARGUMENTWÖRTER

Es kann natürlich sein, daß sowohl die Quelle als auch das Ziel Argumentwörter mit sich führen.

Beispiel: der Befehl

```
MOVE.L #$1235678, $23456789.L
    (Speichere  die  Zahl $23456789 in
    die Speicherstelle $12345678 ab.)
```

hat vier Argumentwörter: zwei, die sich auf die Quelle und zwei, die sich auf das Ziel beziehen.

Adresse			
1000	MOVE.L #<data>,xxx (L)		Befehlswort
1002	12	34	1. Argumentwort der Quelle
1004	56	78	2. Argumentwort der Quelle
1006	23	45	1. Argumentwort des Ziels
1008	67	89	2. Argumentwort des Ziels
100A	ADD.L D2, D5		nächstes Befehlswort

Im ersten Argumentwort ist die höherwertige Hälfte des Quell-Arguments vorhanden, im zweiten Argumentwort die niederwertige Hälfte des Quell-Arguments, im dritten die höherwertige Hälfte des Ziel-Arguments, und im vierten Argumentwort die niederwertige Hälfte des Ziel-Arguments.

An dem Befehlswort "sieht" der Prozessor, daß die vier Wörter (in diesem Beispiel), die dem Befehlswort folgen, als Argument zu nehmen sind.

Das dem letzten Argumentwort folgende Wort wird als nächstes Befehlswort genommen.

3.8.5 ZUSAMMENFASSUNG

Im Befehlswort ist die Information enthalten, ob dem Befehl keine, ein oder zwei Argumentwörter folgen.

Der Prozessor nimmt dementsprechend dieses Wort, das dem Befehl oder dem letzten Argumentwort folgt, als Befehlswort: zuerst auf die Quelle, und dann auf das Ziel bezogen.

Kapitel 4

Befehlssatz

4.1 REIHENFOLGE DER PARAMETER

Es gibt Befehle ohne Parameter oder mit einem oder mit zwei Parametern.

Die Syntax bei den Befehlen mit zwei Parameter hat einheitlich die Reihenfolge "von links nach rechts":

```
<Befehl>   <Quelle>,
```

So kopiert der Befehl

```
MOVE.L D3,D5
```

die Daten VON Register D3 NACH Register D5.

Programmierer, die früher auf anderen Prozessoren programmiert haben, kennen unter Umständen Assembler-Befehle, die "von links nach rechts" gehen, wie

```
STORE <Quelle> <Ziel>
```

und andere Assembler-Befehle, die "von rechts nach links" gehen, wie:

```
LOAD <Ziel> <Quelle>
```

Sie werden sich freuen, weil diese beiden Befehle durch einen neuen Befehl abgedeckt sind und daß alle Befehle zumindest in gleiche "Richtung" gehen.

4.2 KLASSEN DER BEFEHLE

Für den 68000-Programmierer gibt es die folgenden Befehlsklassen:

1. Daten-Kopierbefehle
2. Integer arithmetische Befehle
3. Logische Befehle
4. Schiebe- und Rotierbefehle
5. Bit Befehle
6. BCD-Befehle
7. Programmsteuerbefehle
8. Systemsteuerbefehle
9. Multiprozessor Befehle

Die Befehle werden hier kurz vorgestellt. Eine komplette Beschreibung sämtlicher Befehle ist in Anhang B enthalten.

4.2.1 DATEN-KOPIERBEFEHLE

EXG	Vertausche Register
LEA	Lade effektive Adresse im Register
LINK	Reserviere Bereich im Stack
MOVE, MOVEA	Kopiere Daten
MOVEM	Kopiere mehrere Register
MOVEP	Eingabe/Ausgabe zur/von Peripherie
MOVEQ	kopiere Konstante "quick" (8-Bit)
PEA	Lege effektive Adresse im Stack ab
UNLK	Löse Reservierung im Stack auf

4.2.2 INTEGER ARITHMETISCHE BEFEHLE

ADD, ADDA	Addiere binär
ADDI, ADDQ	Addiere Konstante
ADDX	Addiere mit Extend-Bit
CLR	Lösche Operand
CMP, CMPA	Vergleiche
CMPI	Vergleiche Konstante
CMPM	Vergleiche Speicherinhalt
DIVS, DIVU	Dividiere ohne Vorzeichen
EXT	Erweitere Vorzeichen
MULS, MULU	Multipliziere
NEG	Negiere Operand
NEGX	Negiere Operand mit Extend-Bit
SUB, SUBA	Subtrahiere binär
SUBI, SUBQ	Subtrahiere Konstante
SUBX	Subtrahiere mit Extend-Bit
TST	Prüfe Operand

4.2.3 LOGISCHE BEFEHLE

AND	Logisches UND
ANDI	UND mit Konstante
EOR	Exclusives ODER
EORI	Exclusives ODER mit Konstante
NOT	Logisches Komplement
OR	Logisches ODER
ORI	ODER mit Konstante

4.2.4 SCHIEBE- UND ROTIERBEFEHLE

ASL	Arithmetisches Schieben nach links
ASR	Arithmetisches Schieben nach rechts
LSL	Logisches Schieben nach links
LSR	Logisches Schieben nach rechts
ROL	Rotiere links ohne Extend-Bit
ROR	Rotiere rechts ohne Extend-Bit
ROXL	Rotiere links mit Extend-Bit
ROXR	Rotiere rechts mit Extend-Bit
SWAP	Vertausche Register-Hälften

4.2.5 BIT-BEFEHLE

BCHG	Prüfe Bit und ändere
BCLR	Prüfe Bit und lösche
BSET	Prüfe Bit und setze
BTST	Prüfe Bit
TAS	Prüfe und setze Operand

4.2.6 BCD-BEFEHLE

ABCD	Addiere BCD-Zahl mit Extend-Bit
NBCD	Negiere BCD-Zahl mit Extend-Bit
SBCD	Subtrahiere BCD-Zahl mit Extend-Bit

4.2.7 PROGRAMMSTEUERBEFEHLE

4.2.7.1 BEDINGTE PROGRAMMSTEUERBEFEHLE

Bcc	Springe bedingt
DBcc	prüfe, dekrementiere und springe
Scc	Setze Byte aufgrund Bedingung

Dieses "cc" steht stellvertretend für die entsprechende Bedingung. Für die komplette Befehlsverzeichnis, siehe bei Bcc, DBcc oder Scc nach.

4.2.7.2 UNBEDINGTE PROGRAMMSTEUERBEFEHLE

BRA	Springe unbedingt (relativ)
BSR	Aufruf Unterprogramm (relativ)
JMP	Springe unbedingt
JSR	Aufruf Unterprogramm
NOP	Tue nichts

4.2.7.3 RÜCKKEHRBEFEHLE

RTR	Rückkehr Unterprogramm + Rückladen CCR
RTS	Rückkehr vom Unterprogramm
RTE	Rückkehr von Exception (P)

4.2.8 SYSTEM-STEUERBEFEHLE

4.2.8.1 PRIVILEGIERTE SYSTEM-STEUERBEFEHLE

ANDI zum SR	UND mit Konstante zum SR (P)
EORI zum SR	Exclusives ODER mit Konstante zum SR (P)
MOVE vom SR	Kopiere vom Status Register (P)
MOVE zum SR	Kopiere zum Status Register (P)
MOVE USP	Kopiere vom/zum User Stack Pointer (P)
ORI zum SR	ODER mit Konstante zum SR (P)
RESET	Rücksetzen Eingabe-Ausgabe (P)
STOP	Lade Status Register und Halt (P)
RTE	Rückkehr von Exception (P)

(P) = privilegierter Befehl

4.2.8.2 EXCEPTION AUSLÖSENDE SYSTEM-STEUERBEFEHLE

CHK	Vergleiche Register mit Grenzen
ILLEGAL	Löse Illegal-Exception aus
TRAP	Trap
TRAPV	Trap, wenn Überlauf

Darüber hinaus lösen auch die folgenden Situationen eine Exception aus:

- o jeden nicht erlaubtes Befehl
- o jede nicht erlaubte Adressierungsart
- o Befehle, deren Code mit Axxx oder Fxxx anfangen
- o Verletzung Privilegien
- o Hardware Fehler
- o Interrupts

Für weitere Details siehe Kap 6.2.

4.2.8.2 SYSTEM-STEUERBEFEHLE ZUM CCR

MOVE zum CCR	Kopiere zum Condition Code Register
ANDI zum CCR	UND mit Konstante zum CCR
EORI zum CCR	Exclusives ODER mit Konstante zum CCR
ORI zum CCR	ODER mit Konstante zum CCR

Kapitel 5
Adressierungsarten der Befehle

Die meisten hier oben genannten Befehle haben mehrere Adressierungsarten, d.h., innerhalb eines Befehls kann ich die Operanden in verschiedenen Arten adressieren.

Dadurch, daß so viele Befehle in so vielen Adressierungsarten benutzt werden können, zeigt sich die Vielseitigkeit des 68000 Prozessors. Durch diese Vielzahl der Möglichkeiten ist dieses Kapitel vielleicht das Schwierigste des ganzen Buches geworden.

In der Befehls-übersicht im Anhang B steht bei den meisten Befehlen eine Übersicht der Adressierungsarten, etwa nach diesem Muster.

Adr.-Art	Mode	Reg
Dn	000	R:Dn
An	001	R:An
(An)	010	R:An
(An)+	011	R:An
-(An)	100	R:An
d16(An)	101	R:An
d8(An,Xi)	110	R:An

Adr.-Art	Mode	Reg
xxx.W	111	000
xxx.L	111	001
d16(PC)	111	010
d8(PC,Xi)	111	011
#<data>	111	100
Erläuterung siehe Kapitel 5		

Steht bei einer Adressierungsart der Vermerk "nicht erlaubt", dann bedeutet das nur, daß diese Adressierungsart in dem Befehlssatz des 68000 nicht vorgesehen ist.

(Der entsprechende Maschinenkode ist möglicherweise mit einem anderen Befehl belegt, oder es findet eine Illegal Exception statt.)

5. Adressierungsarten von Befehlen

Die hier erwähnten zwölf Adressierungsarten werden wir hier nacheinander erläutern. Sie sind in numerischer Reihenfolge gegliedert.

> (Manche Befehle nehmen implizit Bezug auf den Programmzähler, den Stack Pointer oder das Status Register. Sie werden in diesem Kapitel nicht besprochen.)

D i e   e f f e k t i v e   A d r e s s e

In den meisten Befehlen wird auf dem Operand mit einer "effektiven Adresse" Bezug genommen. Diese effektive Adresse ist eine Bitkombination innerhalb des Maschinenbefehls, aus dem sich der Operand determinieren läßt. Dieses Kapitel zeigt wie.

Die effektive Adresse ist meistens an der niederwertigen Seite des Befehlswortes abgelegt.

Manche Befehle, wie z.B. MOVE, haben zwei effektive Adressen, eine für den Quelloperanden und eine für den Zieloperanden.

```
  15  14  13  12  11  10  9   8   7   6   5   4   3   2  1  0
|   |   |   |   |   |   |   |   |   |   |   |                    |
| X | X | X | X | X | X | X | X | X | X |  Effektive Adresse |
|   |   |   |   |   |   |   |   |   |   |     Mode   | Register|
```

Die effektive Adresse besteht aus zwei Feldern mit je drei Bit, das Mode-Feld und das Register-Feld. Das Mode-Feld wählt die Adressierungsart an und das Registerfeld bezeichnet meistens ein Register.

Assembeler Syntax:	Adr.-Art	Mode	Reg
Dn (n in 0..7)	Dn	000	R:Dn

Vorgang: Operand = Dn

DATENREGISTER DIREKTE ADRESSIERUNG Dn

Der Operand ist das Datenregister - spezifiziert im Registerfeld. Mit den drei Bits des Registerfeldes werden die Datenregister D0..D7 angewählt.

Datenregister Dn

31 ... 0
O P E R A N D

BEISPIEL:

Befehl	vorher	nachher
MOVE.B D2,D4	D2 = 22222222	D2 = 22222222
	D4 = 44444444	D4 = 44444422
ADD.W D1,D2	D1 = 11111111	D1 = 11111111
	D2 = 22222222	D2 = 11113333
CLR.L D7	D7 = 77777777	D7 = 00000000

Adr.-Art	Mode	Reg
An	001	R:An

Assembeler Syntax:
An (n in 0..7)

Vorgang: Operand = An

5.2 ADRESSREGISTER DIREKTE ADRESSIERUNG An

Der Operand ist. das Adressregister - spezifiziert im Registerfeld. Mit den drei Bits des Registerfeldes werden die Datenregister D0..D7 angewählt.

Adressregister An



Das Verhalten ist fast so wie mit Datenregister direkter Adressierung, mit aber zwei wichtigen Abweichungen:

- o Die Übertragung von Daten über Adressregister ist auf Wörter und Langwörter beschränkt: die Übertragung von Bytes ist nicht erlaubt.
- o Wenn ein Wort in ein Adressregister hineinkopiert wird, wird Bit 15 (das Vorzeichenbit) in den Bitpositionen 31..16 hineingeschrieben.

---

*) Das Symbol *) bei der Befehlserklärung (Anhang B) bedeutet, daß diese Adressierungsart nur für Wort-und Langwort-Befehle, aber nicht für Byte-Befehle erlaubt ist.

Assembler Syntax:
An (n in 0..7)

Adr.-Art	Mode	Reg
An	001	R:An

Vorgang: Operand = An

BEISPIEL:

Befehl	vorher	nachher
MOVE.B A1,D3	A1 = 11111111	A1 = 11111111
	D3 = 33333333	D3 = 33333311
MOVE.B D2,AO	nicht erlaubt *)	
MOVE.W A1,D3	A1 = 11111111	A1 = 11111111
	D3 = 33333333	D3 = 33331111
ADDA.W D2,A4	D2 = 22222222	D2 = 22222222
	A4 = 44444444	A4 = 00006666
ADDA.W D2,A4	D2 = DDDDDDDD	D2 = DDDDDDDD
	A4 = 11111111	A4 = FFFFEEEE
MOVE.L A1,D3	A1 = 11111111	A1 = 11111111
	D3 = 33333333	D3 = 11111111
CLR.L D2	D2 = 22222222	D2 = 00000000

*) Wir haben MOVE als Beispiel genommen.
Byte-Operationen zum Adressregister sind bei MOVE nicht erlaubt (siehe Befehlserklärung von MOVE in Anhang B).

Adr.-Art	Mode	Reg
(An)	**010**	**R:An**

Assembler Syntax:
(An)   (n in 0..7)

Vorgang: Adresse Operand = An

5.3 ADRESSREGISTER INDIREKTE ADRESSIERUNG (An)

Mit den drei Bits des Registerfeldes werden die Adressregister A0..A7 angewählt.

In dem angewählten Adressregister befindet sich die Adresse des Operanden.

Der Operand befindet sich also an der Speicherstelle, wohin das Adressregister zeigt.

```
                              31                    0
Adressregister An            |   SPEICHER ADRESSE   |
                                        ▼
Speicherstelle               |        OPERAND       |
```

Assembler Syntax:
(An)   (n in 0..7)

Adr.-Art	**Mode**	**Reg**
(An)	**010**	**R:An**

Vorgang: Adresse Operand = An

BEISPIEL:

Wenn das Registerfeld die Bits 010 (Wert 2) enthält, und im Adressregister A2 der Wert $1000 abgespeichert ist, dann wird der Wert auf der Adresse $1000 im Speicher als Operand genommen.

(Ob dieser Speicherinhalt nun als Byte, Wort oder Langwort genommen wird, hängt von der Datentyp-Bezeichnung hinter dem Befehl ab, also ".B", ".W" oder ".L". Weitere Details über die Datentypen siehe Kap 3.4)

Befehl	vorher	nachher
MOVE.L (A2),D1	A2 = 00001000	A2 = 00001000
	D1 = 11111111	D1 = 12345678
	auf $00001000 ist 12345678	12345678

Adr.-Art	Mode	Reg
(An)+	011	R:An

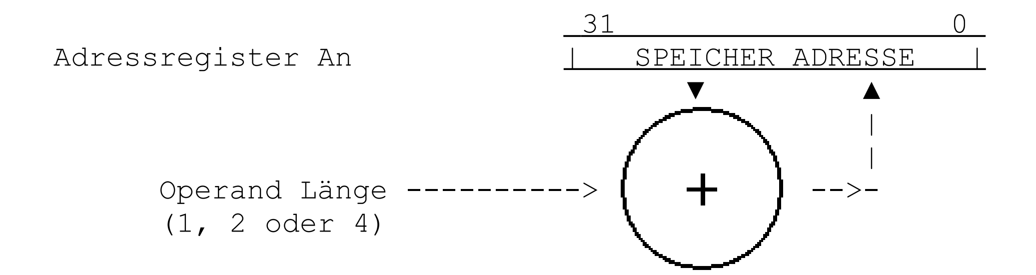
Assembler Syntax:
(An)+ (n in 0..7)

Vorgang: Adresse Operand = An
An=An+K (K=1, 2 oder 4)



5.4 ADRESSREGISTER INDIREKTE ADRESSIERUNG
MIT POST-INKREMENT (An)+

```
                                  31                    0
Adressregister   An              |   SPEICHER  ADRESSE   |
                                            ▼
Speicherstelle                   |         OPERAND       |
31
```



Mit den drei Bits des Registerfeldes werden die Adressregister A0..A7 angewählt.

In dem angewählten Adressregister befindet sich die Adresse des Operanden.

Der Operand befindet sich also an **der** Speicherstelle, wohin das Adressregister zeigt.

Assembler Syntax:
(An)+ (n in 0..7)

Adr.-Art	Mode	Reg
(An)+	011	R:An

Vorgang: Adresse Operand = An
An=An+K (K=1, 2 oder 4)

Bis soweit ist diese Adressierungsart identisch mit der Adressierungsart "Adressregister indirekte Adressierung (An)", wie hier oben besprochen.

Danach nimmt das Adressregister um die Größe des Operanden zu.

Mit dem Befehl MOVE.B (A1)+,D0 wird AI also um eins,
mit dem Befehl MOVE.W (A1)+,D0 um zwei,
mit dem Befehl MOVE.L (A1)+,D0 um vier vergrößert.

BEISPIEL:

Wenn das Registerfeld die Bits 010 (Wert 2) enthält, und im Adressregister A2 der Wert $1000 abgespeichert ist, dann wird der Wert auf der Adresse $1000 im Speicher als Operand genommen.

Erst nach Auswertung des Operanden vergrößert sich der Wert von A2.

Befehl	vorher	nachher
MOVE.L (A2)+,D1	A2 = 00001000	A2 = 00001004
	D1 = 11111111	D1 = 12345678
	auf $00001000 ist 12345678	12345678

Adr.-Art	Mode	Reg
(An)+	011	R:An

Assembler Syntax:
(An)+ (n in 0..7)

Vorgang: Adresse Operand = An
An=An+K (K=1, 2 oder 4)

Das Adressregister A7, der Stack Pointer, verhält sich geringfügig anders als die anderen Adress-register.

Beim Byte-Befehl, z.B.

```
MOVE.B (A7)+,D0
```

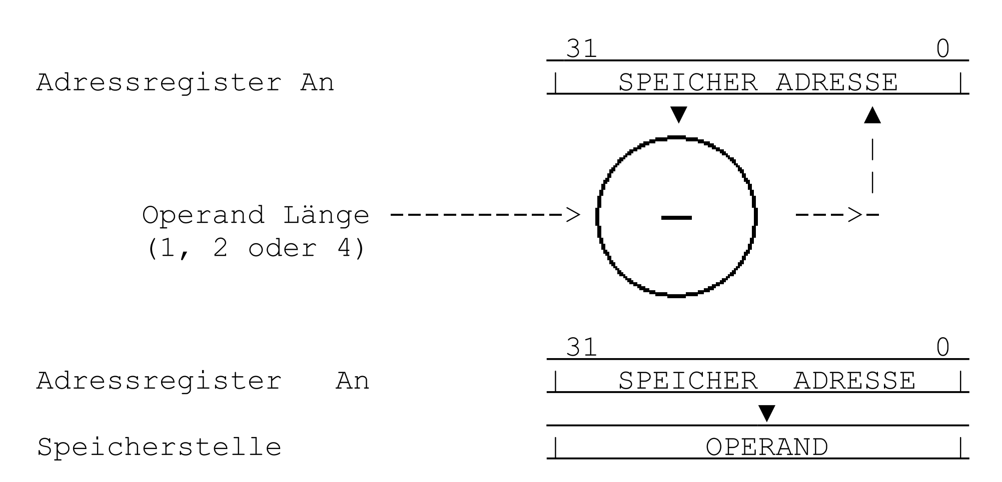
nimmt A7 um zwei statt um eins zu. Damit ist gewährleistet, daß der Stack Pointer immer zu einer geraden Adresse zeigt.

Assembler Syntax:
-(An) (n in 0..7)

Adr.-Art	Mode	Reg
-(An)	100	R:An

Vorgang: An=An-K (K=1, 2 oder 4)
Adresse Operand = An

5.5 ADRESSREGISTER INDIREKTE ADRESSIERUNG MIT PRÄ-DEKREMENT -(An)



Mit den drei Bits des Registerfeldes werden die Adressregister A0..A7 angewählt.

Zuerst wird das Adressregister um die Größe des Operanden vermindert.

Mit dem Befehl MOVE.B -(AI),D0 wird AI also um eins,
mit dem Befehl MOVE.W -(AI),D0 um zwei,
mit dem Befehl MOVE.L -(A1),D0 um vier vermindert.

Nach dieser Verminderung bildet das angewählte Adressregister die effektive Adresse des Operanden.

Adr.-Art	Mode	Reg
-(An)	100	R:An

Assembler Syntax:
-(An) (n in 0..7)

Vorgang: An=An-K (K=1, 2 oder 4)
Adresse Operand = An

Der Operand befindet sich also an der Speicherstelle, wohin das Adressregister zeigt. In dieser Hinsicht ist diese Adressierungsart identisch mit der Adressierungsart "Adressregister indirekte Adressierung (An)", wie hier oben besprochen.

BEISPIEL:

Wenn das Registerfeld die Bits 010 (Wert 2) enthält, und im Adressregister A2 der Wert $1004 abgespeichert ist, dann wird bei der Ausführung des Befehls

```
MOVE.L   -(A2),D1
```

zuerst der Wert von A2 um vier verringert. Der Wert ändert sich von $1004 auf $1000. Danach wird der Wert auf der Adresse $1000 als Operand genommen.

Befehl	vorher	nachher
MOVE.L -(A2) ,D1	A2 = 00001004	A2 = 00001000
	D1 = 11111111	D1 = 12345678
	auf $00001000 ist 12345678	12345678

Assembler Syntax:
-(An) (n in 0..7)

Adr.-Art	Mode	Reg
-(An)	100	R:An

Vorgang: An=An-K (K=1, 2 oder 4)
Adresse Operand = An

Das Adressregister A7, der Stack Pointer, verhält sich geringfügig anders als die anderen Adressregister.

Beim Byte-Befehl, z.B.

```
MOVE.B -(A7),D0
```

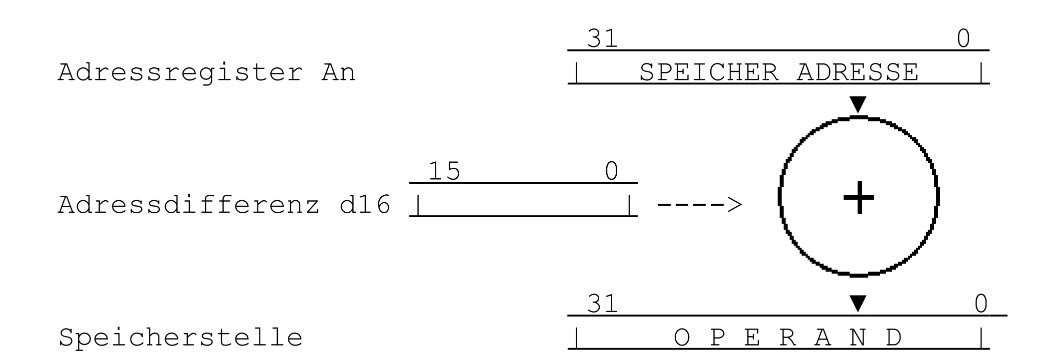
wird A7 um zwei statt um eins verringert. Damit ist gewährleistet, daß der Stack Pointer immer zu einer geraden Adresse zeigt.

Adr.-Art	Mode	Reg
d16(An)	101	R:An

Assembler Syntax:
d16 (An) oder d16,An)
(d16 ist ein 16-Bit Wort, n in 0..7)

Vorgang: Adresse Operand = An+d16

5.6 ADRESSREGISTER INDIREKTE ADRESSIERUNG MIT ADRESSENDIFFERENZ d16(An)



Mit den drei Bits des Registerfeldes werden die Adressregister A0..A7 angewählt.

Die Werte des angewählten Adressregisters und die Konstante werden zusammenaddiert. Das Ergebnis bildet die effektive Adresse des Operanden.

Der Operand befindet sich also an der Speicherstelle, wohin das Adressregister zeigt.

Assembler Syntax:
d16 (An) oder d16,An)
(d16 ist ein 16-Bit Wort,
n in 0..7)

Adr.-Art	Mode	Reg
d16(An)	**101**	**R:An**

Vorgang: Adresse Operand = An+d16

BEISPIEL:

Wenn das Registerfeld die Bits 010 (Wert 2) enthält, und im Adressregister A2 der Wert $1000 abgespeichert ist, dann bezieht sich 8(A2) auf die Speicherstelle $1008.

Umgekehrt bezieht sich -8(A2) auf die Speicherstelle $0FF8.

Diese Adressierart ist ein vielbenutztes Verfahren, um innerhalb von verschiedenen Tabellen schnell die entsprechenden Werte zu adressieren.

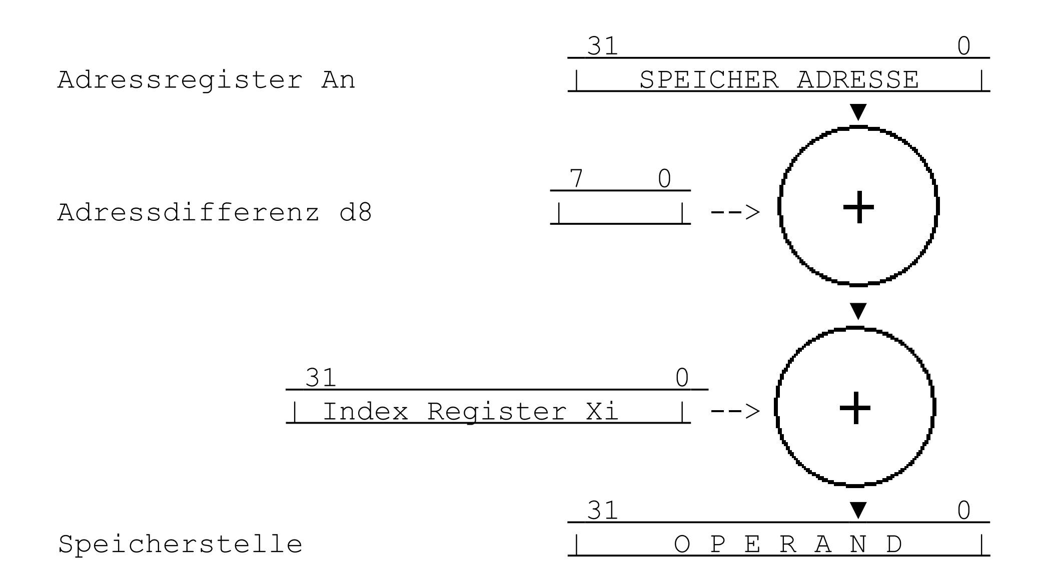
Befehl	vorher	nachher
MOVE.L 8(A2),D1	A2 = 00001000	A2 = 00001000
	D1 = 11111111	D1 = 12345678
	auf $00001008 ist 12345678	12345678

Adr.-Art	Mode	Reg
d8(An,Xi)	110	R:An

Assembler Syntax:
d8(An,Dn.W) oder (d8,An,Dn.W)
d8(An,Dn.L) oder (d8,An,Dn.L)
d8(An,An.W) oder (d8,An,An.W)
d8(An,An.L) oder (d8,An,An.L)
(d8 ist ein 8-Bit Wort,
n in 0..7)

Vorgang:
Adresse Operand=
An+Xi+d8

5.7 ADRESSREGISTER INDIREKTE ADRESSIERUNG MIT INDEX d8(An,Xi)



Assembler Syntax:
d8(An,Dn.W) oder (d8,An,Dn.W)
d8(An,Dn.L) oder (d8,An,Dn.L)
d8(An,An.W) oder (d8,An,An.W)
d8(An,An.L) oder (d8,An,An.L)
(d8 ist ein 8-Bit Wort,
n in 0..7)

Adr.-Art	Mode	Reg
d8(An,Xi)	110	R:An

Vorgang:
Adresse Operand=
An+Xi+d8

Diese Adressierungsmethode ermöglicht es uns, die Adresse des Operanden über gleich drei Parameter anzuwählen:

d8 eine Konstante. Diese Konstante hat eine Größe von 7 Bit plus Vorzeichen.

An ein Adressregister.

Xi den Inhalt eines zweiten Daten- oder Adress-Registers (das Index-Register).
Dem Index-Register wird eine Längen-Angabe mitgegeben. So bedeutet "A2.L", daß sämtliche Bytes des Adressregisters berücksichtigt werden. "D0.W" bedeutet, daß nur die niederwertigen Bits 0..15 des Registers D0 berücksichtigt werden, wobei Bit 15 als Vorzeichenbit genommen wird.

Die Werte des Adressregisters, des Indexregisters und die Konstante werden zusammenaddiert. Das Ergebnis bildet die Adresse des Operanden.

Der Operand befindet sich an der Speicherstelle, wohin der Summenwert zeigt.

Adr.-Art	Mode	Reg
d8(An,Xi)	110	R:An

Vorgang:
Adresse Operand=
An+Xi+d8

Assembler Syntax:
d8(An,Dn.W) oder (d8,An,Dn.W)
d8(An,Dn.L) oder (d8,An,Dn.L)
d8(An,An.W) oder (d8,An,An.W)
d8(An,An.L) oder (d8,An,An.L)
(d8 ist ein 8-Bit Wort,
n in 0..7)

Diese Adressierungsart bietet die Möglichkeit, auch die Datenregister für Adressierungen zu benützen. Das ist ein Vorteil, weil mit Datenregistern mehr Rechenoperationen erlaubt sind als mit Adressregistern.

Sie können auch auf den Einsatz eines Indexwertes verzichten: mit der Adressierung MOVE 0(A2,D0),D1 benutzen Sie dann als Adresse die Summe zweier Register.

BEISPIEL:

Wenn das Registerfeld die Bits 010 (Wert 2) enthält, im Adressregister A2 der Wert $1000 abgespeichert ist und das Datenregister D0 den Wert 10 enthält, dann bezieht sich 8(A2, D0) auf die Speicherstelle $1018.

Befehl	vorher	nachher
MOVE.L 8(A2,DO),D1	A2 = 00001000	A2 = 00001000
	D0 = 00000010	D0 = 00000010
	D1 = 11111111	D1 = 12345678
	auf $00001018 ist 12345678	12345678

Assembler Syntax:
d8(An,Dn.W) oder (d8,An,Dn.W)
d8(An,Dn.L) oder (d8,An,Dn.L)
d8(An,An.W) oder (d8,An,An.W)
d8(An,An.L) oder (d8,An,An.L)
(d8 ist ein 8-Bit Wort,
n in 0..7)

Adr.-Art	Mode	Reg
d8(An,Xi)	110	R:An

Vorgang:
Adresse Operand=
An+Xi+d8

Um die Information des Index-Registers im Computer unterbringen zu können, brauchen wir nach dem Befehlswort ein Argumentwort - und zwar mit dem folgenden Format:

```
 15  14  13  12  11  10  9   8   7   6   5   4   3   2  1  0
|   |           |   |   |   |   |                            |
|D/A| Register  |W/L| 0 | 0 | 0 |     I   N   D   E   X      |
|___|___________|___|___|___|___|____________________________|
```

Bit 15 besagt, ob das Index-Register in Bit 14.. 12 ein Datenregister oder ein Adressregister ist.
0 - Datenregister
1 - Adressregister

Bit 14..12 ist die Nummer des Daten- oder Adressregisters.

Bit 11 Größe des Index-Registers
0 - Nur die minderwertigen Bits 0..15 des Index-Registers werden benutzt, wobei Bit 15 als Vorzeichen genommen wird.
1 - Sämtliche Bits des Index-Register werden berücksichtigt.

Adr.-Art	Mode	Reg
xxx.W	111	000

Assembler Syntax:
xxx.W
(xxx ist eine 16-Bit Adresse)

Vorgang: Adresse Operand = xxx.W

5.8 ABSOLUTE KURZE ADRESSIERUNG (xxx.W)

Die Bitkombination 000 im Registerfeld wählt nicht ein Register an, er dient vielmehr dazu, um - zusammen mit dem Wert im Mode-Feld - die Adressierungsart festzulegen.

Der Wort-Operand adressiert eine 16 Bit Adresse im Speicher. Dabei wird Bit 15 als Vorzeichen benutzt.
Die Werte $0000..$7FFF adressieren daher die Speicherstellen $00000000 bis $00007FFF, also die ersten 32 kByte des Speicherbereichs.

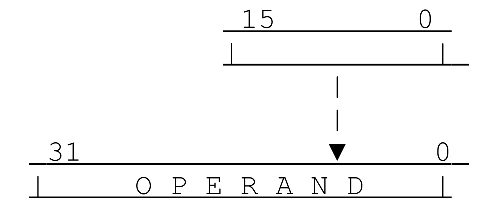
> (Die Werte $8000 bis $FFFF adressieren die Speicherstellen $FFFF8000 bis $FFFFFFFF. Da bei dem 68000 die höchstwertigen 8 Adressbits nicht ausgewertet werden, ist der Einsatz dieser Werte nicht sinnvoll.)

Mit der Adressierungsart "absolute lange Adressierung" (nächster Abschnitt) können Sie sämtliche Speicherplätze absolut adressieren.

Assembler Syntax:
xxx.W
(xxx ist eine
16-Bit Adresse)

Adr.-Art	Mode	Reg
xxx.W	111	000

Vorgang: Adresse Operand = xxx.W



BEISPIEL: Der Befehl

```
MOVE.L $1111.W, D1
```

kopiert den Inhalt der Speicherstelle $00001111 in das Datenregister D1.

> (Zur Erinnerung: an der Erweiterung ".L" hinter "MOVE" im Beispiel sehen wir, daß vier Byte kopiert werden, siehe Kap. 3.4. Die Anordnung dieser Bytes wird in Kap. 3.5 erläutert.)

Adr.-Art	Mode	Reg
xxx.W	111	000

Assembler Syntax:
xxx.W
(xxx ist eine 16-Bit Adresse)

Vorgang: Adresse Operand = xxx.W

Als Beispiel zeigen wir Ihnen die Anordnung des Befehls, wobei der MOVE-Befehl im ersten Wort und das Argument $1111 im zweiten Wort steht. Für weitere Details des MOVE-Befehls siehe Anhang B.

Die Begriffe "Argument" und "Argumentwort" werden in Kap. 3.8 erläutert.

```
| 15  14  13  12| 11  10  9   8   7   6   5   4   3   2   1   0
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 |
|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|
  M O V E . L   | <- reg -> | <- mode ->|<- mode -> | <- reg -> |
  ($21C1)       | <-  Z   i   e   l   ->|<- Q  u  e  l  l  e  ->|

_________________________________________________________________
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|
                |               |               |
 ($1111)
```

```
Befehlswort         $21C1   MOVE.L $1111.W, D1
Argumentwort        $1111
```

Assembler Syntax:
xxx.L
(xxx ist eine
32-Bit Adresse)

Adr.-Art	Mode	Reg
xxx.L	111	001

Vorgang: Adresse Operand = xxx.L

5.9 ABSOLUTE LANGE ADRESSIERUNG (xxx.L)

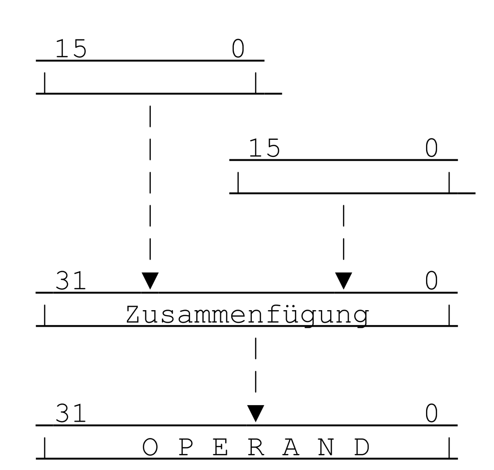
Die Bitkombination 001 im Registerfeld wählt nicht ein Register an, er dient vielmehr dazu, um - zusammen mit dem Wert im Mode-Feld - die Adressierungsart festzulegen.

Mit der Adressierungsart "absolute lange Adressierung" kann ich - im Gegensatz zu der Adressierungsart "absolute kurze Adressierung" (siehe vorherigen Abschnitt) jede Speicherstelle direkt ansprechen. Sie braucht aber geringfügig mehr Zeit und Speicherplatz als die Adressierungsart "absolute kurze Adressierung".

1. Argumentwort

2. Argumentwort

Speicherstelle



Adr.-Art	Mode	Reg
xxx.L	111	001

Assembler Syntax:
xxx.L
(xxx ist eine 32-Bit Adresse)

Vorgang: Adresse Operand = xxx.L

BEISPIEL: Der Befehl

```
MOVE.L $12345678.L, D1
```

kopiert den Inhalt der Speicherstelle $12345678 in das Datenregister D1.

(Zur Erinnerung: an der Erweiterung ".L" hinter "MOVE" im Beispiel sehen wir, daß vier Byte kopiert werden, siehe Kap. 3.4. Die Anordnung dieser Bytes wird in Kap. 3.5 erläutert.)

Assembler Syntax:
xxx.L
(xxx ist eine
32-Bit Adresse)

Adr.-Art	Mode	Reg
xxx.L	111	001

Vorgang: Adresse Operand = xxx.L

Als Beispiel zeigen wir Ihnen die Anordnung des Befehls, wobei der MOVE-Befehl im ersten, die hochwertige Hälfte der Adresse im zweiten und die niederwertige Hälfte der Adresse im dritten Wort steht. Für weitere Details des MOVE-Befehls siehe Anhang B.

```
| 15  14  13  12| 11  10  9   8   7   6   5   4   3   2   1   0
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 |
|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|
  M O V E . L   | <- reg -> | <- mode ->|<- mode -> | <- reg -> |
  ($23C1)       | <-  Z   i   e   l   ->|<- Q  u  e  l  l  e  ->|
```

```
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0   1 | 0 | 0 |
|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|
                |               |               |
 ($1234)
```

```
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |
|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|
                |               |               |
 ($5678)
```

```
Befehlswort         $23C1      MOVE.L $12345678.W, D1
1. Argumentwort     $1234
2. Argumentwort     $5678
```

Adr.-Art	Mode	Reg
d16(PC)	111	010

Assembler Syntax:
d16(PC) oder (d16,PC)

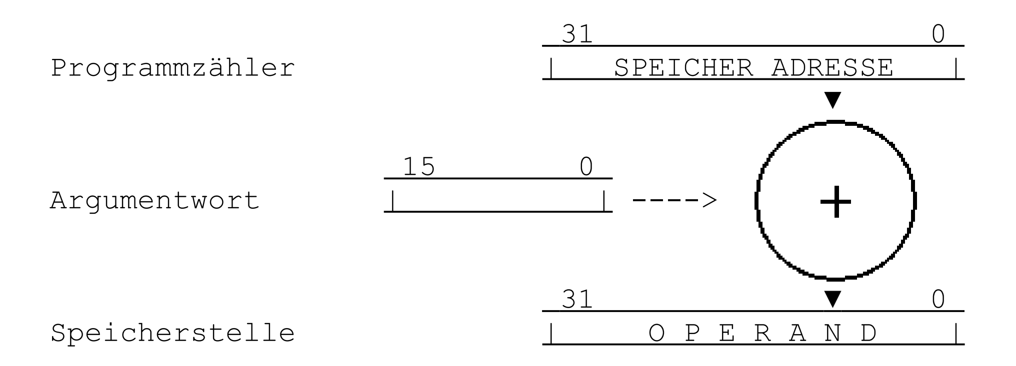
Vorgang: Adresse Operand = PC + d16

5.10 PROGRAMMZÄHLER MIT ADRESSENDIFFERENZ d16(PC)

Die Bitkombination 010 im Registerfeld wählt nicht ein Register an, er dient vielmehr dazu, um - zusammen mit dem Wert im Mode-Feld - die Adressierungsart festzulegen.

Hinter dem Befehlswort kommt im Speicher ein Argumentwort. Das Argumentwort ist ein 16-Bit Wort, wo das Bit 15 als Vorzeichenbit aufgefaßt wird.

Die Adressdifferenz zwischen Operanden und Argumentwort wird im Argumentwort festgehalten.



Assembler Syntax:
d16(PC) oder (d16,PC)

Adr.-Art	Mode	Reg
d16(PC)	111	010

Vorgang: Adresse Operand = PC + d16

BEISPIEL:
Stellen Sie sich im Programm den folgenden Abschnitt vor:

```
  1000 23C1  MOVE.L 20(PC), D1   ; Befehl
  1002 0020

  1022 1234  DC.L   $12345678    ; Argumentwort
  1024 5678
```

Das Argumentwort liegt auf der Adresse 1002. Der Inhalt des Argumentwortes wird zu seiner Adresse addiert. Das Ergebnis ist 1002 + 20 = 1022. Der Operand wird also auf der Adresse 1022 gesucht.

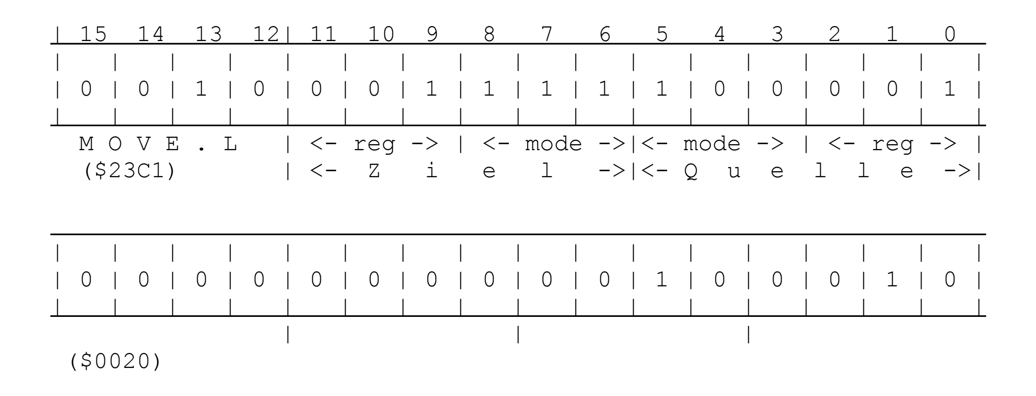
Da es sich in diesem Beispiel um ein langes MOVE handelt, wird ein Langwort auf 1022 gesucht, wobei der hochwertige Teil auf 1022, und der niederwertige Teil auf 1024 liegt.

Befehl	vorher	nachher
MOVE.L 20(PC),D1	D1 = 11111111	D1 = 12345678
	auf $00001022 ist 12345678	12345678

Adr.-Art	Mode	Reg
d16(PC)	111	010

Assembler Syntax:
d16(PC) oder (d16,PC)

Vorgang: Adresse Operand = PC + d16




```
1000 23C1 move.l 20(PC), D1    ; Befehl
1002 0020                      ; Argumentwort
```

Assembler Syntax:	Adr.-Art	Mode	Reg
d16(PC) oder (d16,PC)	d16(PC)	111	010

Vorgang: Adresse Operand = PC + d16

relokatierbares Programmieren

Das interessante bei dieser Adressierungsart ist, daß der Programmcode nicht auf der absoluten Lage des Programms im Speicher Bezug nimmt, sondern auf der Adressdifferenz.

Das bedeutet, das, wenn ich das Programm nicht auf der Adresse $1000, sondern auf der Adresse $2000 laden würde, der Programmcode genau der gleiche wäre.

Auch die Sprünge innerhalb des Programms können wir unabhängig von der Lage im Speicher machen, und zwar mit einem der Branch-Befehle.

Ich könnte also das Programm byteweise zu jeder beliebigen Speicherstelle hin kopieren, es wäre ohne Änderungen ablauffähig.

So ein Programm, das unabhängig von der Speicherlage ablauffähig ist, nennen wie relokatierbar bzw. relocatable.

Für manche Programme ist es vorteilhaft, wenn sie relokatierbar geschrieben sind.

Adr.-Art	Mode	Reg
d8(PC,Xi)	111	011

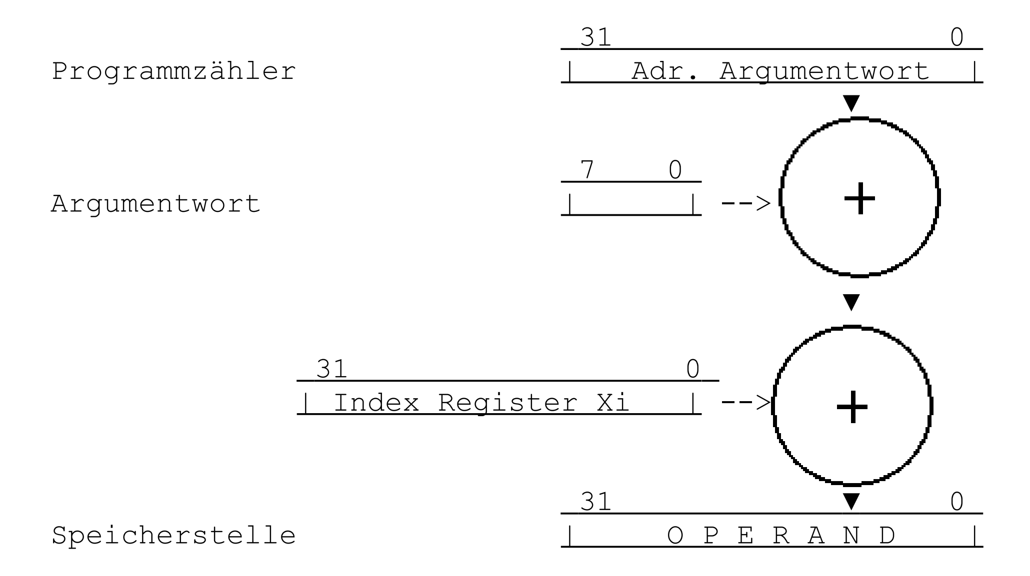
Vorgang:
Adresse Operand=
PC+Xi+d8

Assembler Syntax:
d8(PC,Dn.W) oder (d8,PC,Dn.W)
d8(PC,Dn.L) oder (d8,PC,Dn.L)
d8(PC,An.W) oder (d8,PC,An.W)
d8(PC,An.L) oder (d8,PC,An.L)
(d8 ist ein 8-Bit Wort,
n in 0..7)

PROGRAMMZÄHLER MIT INDEX (d8,PC,Xi)

Die Bitkombination 011 im Registerfeld wählt nicht ein Register an, er dient vielmehr dazu, um zusammen mit dem Wert im Mode-Feld - die Adressierungsart festzulegen.

Sie können aber ein Register angeben. Die Summe dieses Registers, des Programmzählers (PC) und einer Konstanten bilden die Adresse des Operanden.



Adr.-Art	Mode	Reg
d8(PC,Xi)	111	011

Assembler Syntax:
d8(PC,Dn.W) oder (d8,PC,Dn.W)
d8(PC,Dn.L) oder (d8,PC,Dn.L)
d8(PC,An.W) oder (d8,PC,An.W)
d8(PC,An.L) oder (d8,PC,An.L)
(d8 ist ein 8-Bit Wort,
n in 0..7)

Vorgang:
Adresse Operand=
PC+Xi+d8

Diese Adressierungsart ist ähnlich der Adressierungsart mit Adressendifferenz. Sie gibt Ihnen aber zusätzlich die Möglichkeit, über ein Register - das sowohl ein Daten- als auch ein Adressregister sein darf - die Adressierung des Operanden mit zu beeinflussen.

Die Vorteile des relokatierbaren Programmierens - wie bei der Adressierungsart mit Adressdifferenz besprochen - sind hier auch voll nutzbar.

Sie können auch auf den Einsatz eines Indexwert verzichten: mit der Adressierung

```
MOVE 0(PC,A1) ,D1
```

wird die Adresse des Operanden errechnet: aus der Summe des Programmzählers und eines Registers.

Adr.-Art	Mode	Reg
d8(PC,Xi)	111	011

Assembler Syntax:
d8(PC,Dn.W) oder (d8,PC,Dn.W)
d8(PC,Dn.L) oder (d8,PC,Dn.L)
d8(PC,An.W) oder (d8,PC,An.W)
d8(PC,An.L) oder (d8,PC,An.L)
(d8 ist ein 8-Bit Wort,
n in 0..7)

Vorgang:
Adresse Operand=
PC+Xi+d8

BEISPIEL:
Stellen Sie sich im Programm den folgenden Abschnitt vor:

```
          ; Register A1 hat die Wert $2000
  1000 27C1  MOVE.L  8(A1, PC), D1   ; Befehl
  1002 9898                          ; Argumentwort

  1022 1234  DC.L   $12345678        ; hier liegt Operand
  1024 5678
```

Das Argumentwort liegt auf der Adresse 1002. Zu seiner Adresse wird Register AI und der Indexwert addiert. Das Ergebnis ist 1002 + 2000 + 8 = 300A. Der Operand wird also auf der Adresse 300A gesucht.

Befehl	vorher	nachher
MOVE.L 8(PC,A1),D1	A1 = 00002000	A1 = 00002000
	D1 = 11111111	D1 = 12345678
	auf $0000300A ist 12345678	12345678

```
Assembler Syntax:                        |Adr.-Art | Mode|  Reg  |
d8(PC,Dn.W) oder (d8,PC,Dn.W)            |d8(PC,Xi)| 111 |  011  |
d8(PC,Dn.L) oder (d8,PC,Dn.L)
d8(PC,An.W) oder (d8,PC,An.W)
d8(PC,An.L) oder (d8,PC,An.L)                  Vorgang:
(d8 ist ein 8-Bit Wort,                        Adresse Operand=
n in 0..7)                                     PC+Xi+d8
```

Um die Information des Indexregisters im Computer unterbringen zu können, brauchen wir nach dem Befehlswort ein zweites Argumentwort - und zwar mit dem folgenden Format:

```
  15  14  13  12  11  10  9   8   7   6   5   4   3   2  1  0
|   |            |   |   |   |   |                           |
|D/A| Register   |W/L| 0 | 0 | 0 |    I   N   D   E   X      |
|___|____________|___|___|___|___|___________________________|
```

Bit 15 besagt, ob das Index-Register in Bit 14..12 ein Datenregister oder ein Adressregister ist.
- 0 - Datenregister
- 1 - Adressregister

Bit 14.. 12 ist die Nummer des Daten- oder Adressregisters.

Bit 11 Größe des Index-Registers
- 0 - Nur die minderwertigen Bits 0..15 des Index-Registers werden benutzt, wobei Bit 15 als Vorzeichen genommen wird.
- 1 - Sämtliche Bits des Index-Register werden berücksichtigt.

Adr.-Art	Mode	Reg
d8(PC,Xi)	111	011

Assembler Syntax:
d8(PC,Dn.W) oder (d8,PC,Dn.W)
d8(PC,Dn.L) oder (d8,PC,Dn.L)
d8(PC,An.W) oder (d8,PC,An.W)
d8(PC,An.L) oder (d8,PC,An.L)
(d8 ist ein 8-Bit Wort,
n in 0..7)

Vorgang:
Adresse Operand=
PC+Xi+d8

```
| 15  14  13  12| 11  10   9   8   7   6   5   4   3   2   1   0
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 |
|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|
  M O V E . L   | <- reg -> | <- mode ->|<- mode -> | <- reg -> |
  ($27C1)       | <-  Z   i   e   l   ->|<- Q  u   e  l  l  e  ->|

_________________________________________________________________
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|
|D/A| <- reg -> |W/L  0   0   0 | <---  i   n   d   e   x   ---> |
 ($9809)
```

```
1000 27C1  MOVE.L  8(A1, PC), D1   ; Befehl
1002 9808                          ; Argumentwort
```

Assembler Syntax:
#xxxx
(xxxx ist ein 8- 16- oder 32-Bit-Wort)

Adr.-Art	Mode	Reg
#<data>	111	011

Vorgang: Operand = xxxx

5.12 KONSTANTE (UNMITTELBARE DATEN) #<data>

Die Bitkombination 100 im Registerfeld wählt nicht ein Register an, er dient vielmehr dazu, um zusammen mit dem Wert im Mode-Feld - die Adressierungsart festzulegen.

Mit dieser Adressierungsart werden diese Daten, die im Speicher unmittelbar dem Befehlswort folgen, als Operand genommen.

Wenn Sie schön anständig programmieren und keine selbst-modifizierende Programme schreiben, dann können Sie diese Daten also als Konstante betrachten.

BEISPIEL: Der Befehl

```
MOVE.L #$12345678, D1
```

kopiert den Wert $12345678 in das Register D1.

Befehl	vorher	nachher
MOVE.L #12345678, D1	D1=11111111	D1=12345678

Adr.-Art	Mode	Reg
#<data>	111	011

Assembler Syntax:
#xxxx
(xxxx ist ein 8- 16- oder 32-Bit-Wort)

Vorgang: Operand = xxxx

Noch zwei Bemerkungen:

1. Passen Sie hier auf, daß Sie nicht versehentlich das "#"-Zeichen vergessen, denn der Befehl

```
MOVE.L $12345678, D1
```

   kopiert den **Inhalt** der **Speicherstelle** 12345678 in das Register D1, und das ist ganz etwas anderes.

2. Ein Konstante ist nur als **Quelle**, nicht als Ziel einsetzbar. Logisch, denn ein Befehl wie

```
MOVE.L D1, #$12345678
```

   macht keinen Sinn.

Assembler Syntax:
#xxxx
(xxxx ist ein 8- 16- oder 32-Bit-Wort)

Adr.-Art	Mode	Reg
#<data>	111	011

Vorgang: Operand = xxxx

Die Länge-Information des Operanden (Byte, Wort oder Langwort) ist im Befehlswort vermerkt.

Die Argumentwörter sind hier angegeben:

```
| 15  14  13  12| 11  10  9   8 | 7   6   5   4 | 3   2   1   0 |
|   |   |   |   |   |   |   |   |                               |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |           B y t e             |
|___|___|___|___|___|___|___|___|_______________________________|


 _______________________________________________________________
|                                                               |
|                           W o r t                             |
|_______________________________________________________________|


 _______________________________________________________________
|              L a n g w o r t                                  |
|                                                               |
| _ _ _ _ _ _ _ _ _ _ _ _ _ _  hochwertiges Wort _              |
|                              niederwertiges Wort              |
|                                                               |
|_______________________________________________________________|
```

5. Adressierungsarten von Befehlen

Kapitel 6
Stacks, Exceptions und Interrupts

6.1 DER STACK

6.1.1 EINFÜHRUNG

Hier ist wieder ein neuer Begriff für den Assembler-Programmierer .

Ein "Stack" wird auf Deutsch manchmal auch als "Stapel" bezeichnet.

Ein Stack ist ein Speicherverfahren. Dieses Verfahren erlaubt dem Programmierer, Daten, die innerhalb eines Programms anfallen, "automatisch" abzuspeichern und wieder hervorzurufen.

> Das Abspeichern von Daten in ein Stack wird als "push" bezeichnet (to push = schieben).
>
> Das Hervorrufen von Daten aus einem Stack wird als "pop" bezeichnet (to pop = schnell aufschreiben).

Wie wir in den vorherigen Kapiteln gesehen haben, spielt das Adressregister A7 eine Sonderrolle, weil es speziell als Stack Pointer (Pointer = Zeiger) gedacht ist. Manche Befehle nehmen implizit Bezug auf den Stack Pointer A7.

Die Bezeichnung A7 und SP sind für den Assembler identisch. Da wir in diesem Kapitel betonen möchten, daß es sich um den Stack Pointer handelt, wird er hier mit SP bezeichnet.

Es wäre unnötig und verwirrend, wenn Sie ein anderes Register als A7 als Stack Pointer benutzen würden.

Es gibt übrigens zwei verschiedene Stack Pointer A7.

- o Der USP = User Stack Pointer. Wird benutzt wenn der Prozessor in User Mode ist.
- o Der SSP = System Stack Pointer. Wird betätigt, wenn der Prozessor in Supervisor Mode ist. Siehe Kap. 2

Wenn man Anwendungsprogramme in User Mode schreibt, hat man gar nichts mit dem System Stack Pointer zu tun. Ganz im Gegenteil, man sollte sich normalerweise darauf verlassen können, daß der User Stack Pointer, der das Anwendungsprogramm vom Betriebssystem gestellt bekommt, ausreichend viel Platz bietet.

Vom User Mode aus ist der System Stack Pointer nicht zugänglich. Über einen Trap kann man aber in Supervisor Mode gelangen; der User Stack Pointer ist dann über MOVE USP zugänglich.

6.1.2 DIE ARBEITSWEISE EINES STACKS

6.1.2.1 ABLAGE VON DATEN

Stellen Sie sich den folgenden Programmabschnitt vor

```
MOVE.L  #$00001000, SP     ; Initialisierung SP
MOVE.L  #$01234567, D2     ; Initialisierung D2
MOVE.L  D2, -(SP)          ; push D2 auf Stack
(Routine, der D2 zerstört)
MOVE.L  (SP)+, D2          ; pop D2 vom Stack
```

Wenn Sie eifrig das Kapitel 5 gelesen haben, werden Sie zweifellos einsehen, daß beim Befehl

```
     MOVE.L  02, -(SP)          ; push D2 auf Stack
```

zuerst das Adressregister SP um die Datenlänge vermindert wird. Da die Datenlänge wegen der Angabe ".L" ein Langwort = 4 Bytes ist, erhält SP nunmehr den Wert $1000 - $4 = $0FFC. Danach wird D2 in den Speicherplatz $0FFC kopiert.

> Der Inhalt des Registers D2 wird also auf den Stack "gerettet". Dieser Vorgang wird als "push" bezeichnet.

In Kapitel 3.5 haben wir gesehen, daß die hochwertigen Teile auf der niedrigen Adresse, und die niederwertigen Teile auf der höheren Adresse abgespeichert werden. Also befindet sich das hochwertige Datenteil $0123 auf der Adresse $0FFC, und das niederwertige Teil $4567 auf der Adresse $0FFE.

```
 ______________________________________________________
|                         |                            |
|   Vor dem Befehl        |    Nach  dem Befehl        |
|   MOVE.L D2, -(SP)      |    MOVE. L  D2, -(SP)      |
|_________________________|____________________________|
|                         |                            |
|   D2 = $01234567        |    D2 =  $01234567         |
|   SP = $00001000        |    SP =  $00000FFC         |
|                         |                            |
|         0FFC ....       |    SP ->  0FFC 4567        |
|         0FFE ....       |           0FFE 0123        |
|   SP -> 1000 ....       |           1000 ....        |
|         1002 ....       |           1002 ....        |
|_________________________|____________________________|
```

Nachdem die Routine, die den Inhalt von Register D2 zerstört, beendet ist, möchten wir den alten Inhalt von D2 wieder aus unserem Stack hervorholen.

Beim Befehl

```
     MOVE.L  (SP)+, D2          ; pop D2 vom Stack
```

wird zuerst der Inhalt der Speicherstelle, wohin SP zeigt, in D2 kopiert. Da SP den Wert $0FFC hat, erhält D2 den auf der Adresse $0FFC abgespeicherten Wert. Das ist der Wert $01234567. Danach wird das Adressregister SP um die Datenlänge vergrößert. Da die Datenlänge 4 ist, erhält SP den Wert
$0FFC + $4 = $1000.

Vor dem Befehl MOVE.L (SP)+, D2	Nach dem Befehl MOVE.L (SP)+, D2
D2 = $........ SP = $00000FFC	D2 = $01234567 SP = $00001000
SP -> 0FFC 4567 0FFE 0123 1000 .... 1002	0FFC 4567 0FFE 0123 SP -> 1000 .... 1002 ....

Dieser Vorgang, durch welchen das Register D2 wieder mit dem Wert aus dem Stack geladen wird, bezeichnet man als "pop".

6.1.2.2 ABLAGE VON RÜCKKEHRADRESSEN

Auch Rückkehradressen können im Stack abgelegt werden. Schauen sie sich das folgende Beispiel an:

```
 0100         MOVE.L  #$00001000, SP ;  ini SP
 0106         JSR  SUBR    ; Anruf Unterprogramm
 010C         MOVE D2, D4  ; nächster Befehl

 0200 SUBR:   MOVE D3, D5  ; tue irgend etwas
 0202         RTS          ; Rückkehr
```

Beim Befehl

```
 0106         JSR  SUBR    ; Anruf Unterprogramm
```

wird zuerst der Stack Pointer SP um vier vermindert. SP erhält dann den Wert $1000 - $4 = $0FFC. Danach wird die Adresse des **nächsten** Befehls - also $010C auf den Stack abgelegt. Danach findet einen Sprung zu der Adresse SUBR - also $200 statt.

Vor dem Befehl JSR SUBR	Nach dem Befehl JSR SUBR
PC = $00000106 SP = $00001000 0FFC .... 0FFE .... SP -> 1000 .... 1002	PC = $00000200 SP = $00000FFC SP -> 0FFC 010C 0FFE 0000 1000 .... 1002 ....

Beim Befehl

```
 0202        RTS          ; Rückkehr
```

wird zuerst der Inhalt des Stacks - also der Wert $010C - in den Programmzähler PC geladen, so daß der nächste auszuführende Befehl bei $10C liegt. Danach wird der Stack Pointer um 4 erhöht. Er kommt damit wieder von $0FFC auf $1000.

Vor dem Befehl RTS	Nach dem Befehl RTS
PC = $00000202 SP = $00000FFC SP -> 0FFC 010C 0FFE 0000 1000 .... 1002	PC = $0000010C SP = $00001000 0FFC 010C 0FFE 0000 SP -> 1000 .... 1002 ....

Dieses Verfahren läßt sich erweitern; auf den Stack können viele Daten und Rückkehradressen abgespeichert und wieder zurückgewonnen werden.

6.1.2.3 ACHTUNG: STACKFEHLER

Was kann bei der Programmentwicklung alles falsch laufen, wenn man Stacks benutzt?

1. Die Zahl der PUSHes und POPs sind ungleich. Es wird dann z.B. ein abgespeichertes Datenregister als Rückkehradresse genommen. Der Rechner läuft dann "im Wald".

2. Zuviele PUSHes -> Stack Überlauf. Es werden dann Programmteile oder Daten mit Stackdaten überschrieben .

3. Zuviele POPs -> Stack Unterlauf. Es wird jenseits des erlaubten Stackbereichs gelesen. Die letzte Rückkehradresse befindet sich mit Sicherheit nicht dort.

Diese Fehler führen mit großer Wahrscheinlichkeit zum Programmabsturz. Die Erfahrung hat gezeigt, das diese Art von Programmfehlern besonders hartnäckig und schwer zu orten sind. Die Verfasser dieses Werkes haben bereits mehrere graue Haare durch solche Fehler zu verdanken.

6.1.3 WICHTIGE EIGENSCHAFTEN DES STACKS

- Der Stack ist ein LIFO-Speicher (Last In - First Out): die Daten, die Sie zuletzt abgespeichert haben, verlassen den Stack zuerst.
  Der Stack verhält sich damit genauso wie das Papierkörbchen auf Ihrem Schreibtisch: das Dokument, das Sie zuletzt in Ihr Körbchen gelegt haben, entnehmen Sie zuerst. Nur wenn Sie in Ihrem Körbchen wühlen, stimmt der Vergleich nicht.
  Sie müssen also Ihre Daten in umgekehrter Reihenfolge entnehmen, als daß Sie sie angeliefert haben.

- Der Stack wächst bei abnehmendem Stack Pointer, und schrumpft bei zunehmendem Stack Pointer.

6.1.4 DATENLÄNGE AUF DEM STACK

Sie können pro Register unterschiedliche Datenlängen im Stack ablegen.

- Bei einer Datenlänge von 4 Bytes ändert sich der Stack Pointer - wie wir oben gesehen haben, um 4.

- Bei einer Datenlänge von 2 Bytes ändert sich der Stack Pointer erwartungsgemäß um 2.

- Bei einer Datenlänge von einem Byte ändert sich aber der Stack Pointer nicht um 1, sondern um 2, wie in Kap. 5 erwähnt bei den Adressierungen mit Post-Inkrement und Prä-Dekrement. Hiermit wird verhindert, daß der Stack Pointer etwa unbeabsichtigt auf eine ungerade Adresse zeigt.

6.1.5 BEFEHLS-ÜBERSICHT DER STACK-HANDHABUNG

Wie wir oben gesehen haben, können sie mit z.B.

MOVE D2, -(SP) Daten ablegen, und mit
MOVE (SP)+, D2 Daten zurücklesen.

Sie können Register z.B. abspeichern mit

MOVEM.L D0-D7/A0-A6, -(SP) und zurückholen mit
MOVEM.L (SP)+r D0-D7/A0-A6.

Darüber hinaus stehen für die Handhabung des Stack noch die folgenden Befehle zur Verfügung:

JSR Aufruf Unterprogramm (absolut)
BSR Aufruf Unterprogramm (relativ)

RTS Rückkehr vom Unterprogramm
RTR Rückkehr Unterprogramm + Rückladen CCR
RTE Rückkehr von Exception

PEA Lege effektive Adresse im Stack ab
LINK Reserviere Bereich im Stack
UNLK löse Reservierung im Stack auf

6.2 EXCEPTIONS

Eine Exception (= Ausnahme) ist ein Vorgang, indem die normale Abarbeitung der Befehle abgebrochen wird und der Prozessor etwas anderes ausführen soll. Was dann passiert, werden wir in diesem Kapitel besprechen.

Es gibt verschiedene Ursachen für Exceptions.

- ein erwünschtes externes (hardwaremäßiges) Ereignis, auf den der Rechner gewartet hat, z.B. wurde auf der Tastatur ein Zeichen eingegeben. So ein Ereignis bezeichnen wir als **Interrupt**.

- ein unerwünschtes externes (hardwaremäßiges) Ereignis, das evtl. als Fehler zu werten ist, z.B. ein Zugriff auf einer nicht existierenden Speichersteile.

- ein im Programm beabsichtigter Vorgang. Wir haben diese Exception aus programmtechnischen Gründen gewollt.

- ein im Programm möglich auftretender Vorgang, der abgefangen werden muß, z.B. Überschreiten von Grenzen oder Division durch Null.

In dem Prozessor 68000 lösen die folgenden Ereignisse eine Exception aus:

- o Die Befehle TRAF und ILLEGAL lösen **immer** eine Exception aus, ebenso die nicht erlaubten Befehle und Adressierungsarten.
- o Die Befehle TRAPV, CHK, DIVS und DIVU **können** - je nach Ergebnis - eine Exception auslösen.
- o Adressierungsfehler, wie Zugriffe auf einer nicht existenten Speicherstelle oder Zugriff auf ein Wort oder Langwort auf einer ungeraden Speicherstelle, lösen eine Exception aus.
- o Interrupts von Eingabe-Ausgabe-ICs (wenn sie entsprechend angeschlossen sind).

Es findet dann ein Sprung zu einer bestimmten Adresse statt. An dieser Stelle wird die Exception-Behandlungsroutine erwartet.

Der Prozessor findet diese Adresse in der entsprechenden Speicherstelle gemäß der Exceptiontabelle siehe Kap 6.2.2.

6.2.1 WAS PASSIERT BEI EINER EXCEPTION?

1. Zuerst werden das Status Register und der Programmzähler gerettet. Sie werden auf den System Stack gepushed.

Auf dem Stack befinden sich also der Inhalt des Status Registers, wie er gerade vor Eintreten der Exception war, und die Adresse des nächsten Befehls im Speicher.

> Die Adresse des nächsten Befehls ist also die Adresse von diesem Befehl, der im Speicher direkt auf den Befehl folgt, bei dem die Exception ausgelöst wurde. (Siehe auch Kap 6.1.2.2 - Ablage von Rückkehradressen)

(Natürlich liegt der System Stack Pointer nicht auf $1000. Wir benutzen aber gerne konkrete Adressen in unseren Beispielen, um damit irreführende Begriffe wie "oben" und "unten" zu vermeiden.)

Vor der Exception	Nach der Exception
SSP = $00001000	SSP = $00000FFA
0FFA	SSP -> 0FFA Status Register
0FFC	0FFC PC hochw. Wort
0FFE	0FFE PC niederw. Wort
SSP -> 1000	1000 ....
1002	1002 ....

2. Das Status Register wird modifiziert. Das S-Bit wird gesetzt, so daß wir in den Supervisor Mode gelangen. Außerdem wird das T-bit zurückgesetzt, so daß eine evtl. eingeschalteter Trace-Mode während der Exception-Behandlung abgeschaltet wird. Bei einem Interrupt (Externe Exception) wird die Interrupt Maske entsprechend abgeändert.

3. Wenn ein Adressfehler oder ein Busfehler auftritt (Exception 2 bzw. 3), wird zusätzliche Information auf den System Stack gepushed. Für die Beschreibung sehen Sie bitte dort nach.

4. Der Programmzähler PC erhält den Wert, der sich auf der entsprechenden Position in der Tabelle befindet. Der Programmablauf fängt an dieser Speicherstelle an. Hier soll sich die Exception-Behandlungsroutine befinden.

Der Prozessor durchläuft dann die Exception-Behandlungsroutine. Diese Routine wird normalerweise mit ein RTE-Befehl beendet. Es bewirkt, daß das alte Status Register und der alte Programmzähler von dem Stack gepopped werden. Dadurch wird das Programm direkt nach der Stelle fortgesetzt, wo es durch die Exception unterbrochen wurde.

6.2.2 DIE EXCEPTIONTABELLE

Jeder Exception ist eine Vektornummer zugeordnet (0..255).

Im Speicher ist eine Exception-Adresstabelle enthalten. Diese Tabelle fängt bei der . Speicheradresse Null an, und umfaßt 256 Positionen, für jede Exception eine.

Jede Adress-Eintragung (ein Langwort von 32 Bit) belegt in der Tabelle 4 Byte. Die Position innerhalb der Tabelle berechnet sich daher aus (Vektor * 4).

Innerhalb der Tabelle zeigt die Adresse zum der jeweiligen Exception-Behandlungsroutine.

Anfang

BEISPIEL:

> Die Sprungadresse der Exception 5 befindet sich also auf der Speicherstelle 5 * 4 = 20 dezimal, oder $14 hexadezimal. Wenn auf der Adresse $0014 sich die Zahl $1234 und auf $0016 sich die Zahl $5678 befindet, bewirkt die Exception 5 einen Sprung zu der Routine, die auf der Adresse $12345678 anfängt.

Die Exception-Tabelle ist auf der nächsten Seite aufgeführt.

> Jede Exception ist ein Vektor und damit einer Speicheradresse zugeordnet.
>
> An dieser Speicheradresse befindet sich die Startadresse der Exception-Behandlungsroutine.
>
> Diese Routine wird gestartet, wenn die Exception auftritt.

Vektor	Adresse	Zuordnung
0	0000	Reset: Anfangswert SSP
1	0004	Reset: Anfangswert PC
2	0008	Bus Fehler
3	000C	Adressfehler
4	0010	Illegaler Befehl
5	0014	Division durch Null
6	0018	CHK-Befehl
7	001C	TRAPV-Befehl
8	0020	Verletzung Privilegium
9	0024	Trace
10	0028	Line 1010 Emulator
11	002C	Line 1111 Emulator
12-14	0030	) reser-
	bis 0038	) viert
15	003C	nicht initialisiert
16-23	0040	) reser-
	bis 005C	) viert
24	0060	falscher Interrupt
25	0064	Ebene 1 Autovektor Interrupt
26	0068	Ebene 2 Autovektor Interrupt
27	006C	Ebene 3 Autovektor Interrupt
28	0070	Ebene 4 Autovektor Interrupt
29	0074	Ebene 5 Autovektor Interrupt
30	0078	Ebene 6 Autovektor Interrupt
31	007C	Ebene 7 Autovektor Interrupt
32-47	0080	) TRAP-Befehl
	bis 00BC	) 0-15
48-63	00C0	) reser-
	bis 00FF	) viert
64-255	0100	) Anwender
	bis 03FC	) Interrupts 0..191

Bild 6.1 ZUORDNUNG VON EXCEPTIONS

Das Wort "reserviert" in dieser Tabelle bedeutet, daß dieser Vektor durch den Hersteller des 68000 für zukünftige Erweiterungen reserviert ist. Er bittet Sie, diese Vektoren aus Kompatibilitätsgründen nicht zu belegen.

Und warum sollten wir die reservierten Vektoren belegen? Es sind noch so viele Vektoren frei, daß wir nicht das Risiko eingehen sollten, daß unser Programm auf einem zukünftigen 68000 Prozessor (der unter Umständen genauso aussieht) vielleicht nicht mehr läuft.

6.2.3 DIE VERSCHIEDENEN EXCEPTIONS

Jetzt werden wir nacheinander alle Exceptions behandeln, wann sie auftreten und was dann passiert.

Ob eine bestimmte Exception "erwünscht" oder "unerwünscht" ist, liegt im Ermessen des Programmierers. Wir beschreiben hier nur Vorgänge.

Vektor	Adresse	Zuordnung
0	0000	Reset: Anfangswert SSP
1	0004	Reset: Anfangswert PC

Die Exception Null (Reset) tritt auf bei einer Signaländerung auf dem entsprechenden Pin des Prozessor-ICs. Sie hat die höchste Priorität.

Der Reset ist für den Systemstart gedacht. Beim Einschalten, und auch bei Betätigung der Reset-Taste, wird diese Exception aktiviert. Der Programmzähler erhält dabei den Wert, der sich auf der Adresse 0004 befindet. Dieser Wert soll zu der Initialisierungs-routine des Computers zeigen.

Anders als bei den anderen, hiernach zu besprechenden Exceptions, erhält auch der Supervisor Stack Pointer einen Anfangswert, und zwar von der Adresse 0000. Selbstverständlich soll der Zeiger der SSP in RAM (Random Access Memory, Schreib-Lese-Speicher) zeigen!

Weil der alte SSP dabei verloren geht, sind das alte Status Register und der alte Programmzähler nach einer Exception Null nicht mehr verfügbar.

> Die Exception R E S E T ist nicht mit dem B e f e h l R E S E T zu verwechseln. Der Befehl RESET bewirkt lediglich ein hardware-mäßiges Rücksetzen der Eingabe-Ausgabe-ICs, der Programmzähler und der Status bleiben aber beibehalten. Mit dem Befehl RESET kann man vom Programm aus ein hardware Reset durchführen und danach das Programm fortsetzen.

Vektor	Adresse	Zuordnung
2	0008	Bus Fehler

Die Exception 2 tritt auf, wenn der Prozessor versucht, einen Zugriff auf einer nicht existierenden geraden Speicheradresse zu machen. Auf den Stack wird zusätzliche Information über den auftretenden Fehler gepushed. Es hängt von der Exception-Behandlungsroutine ab, ob und wie diese Information ausgewertet wird.

Wenn aber der Prozessor einen Zugriff auf einer nicht existierenden ungeraden Adresse versucht, findet nur die Exception 3 statt, die Exception 2 unterbleibt.

Vor der Exception SSP = $00001000	Nach der Exception SSP = $00000FF2
0FFA	SSP -> 0FF2 Fehlerkode
0FFA	0FF4 Fehler-Adr. h.W.
0FFA	0FF6 Fehler-Adr. n.W.
0FFA	0FF8 Befehlswort
0FFA	0FFA Status Register
0FFC	0FFC PC hochw. Wort
0FFE	0FFE PC niederw. Wort
SSP -> 1000	1000 ....
1002	1002 ....

Auf dem Stack befindet sich ab der Position des Stack Pointers nacheinander die folgende Information:

- Der Fehlerkode. Erklärung siehe weiter unten.
- Hoch- und niederwertiges Teil der Adresse, bei der der Fehler aufgetreten ist.
- Der Inhalt des Befehlswortes, das den Fehler verursachte.
- Status Register zur Fehlerzeit.
- Hoch- und niederwertiges Teil der Adresse des **nächsten** Befehls im Speicher zur Zeit des Fehlers.

F E H L E R K O D E

```
                   4   3   2   1   0
______________________________________
  ░░░|░░░|░░░|░░░|   |   |            |
  ░░░|░░░|░░░|░░░|R/W|I/N|  Zugriff   |     Fehlerkode
__░░░|░░░|░░░|░░░|___|___|____________|
```

R/W = 0 -> Fehler trat auf beim Schreiben
R/W = 1 -> Fehler trat auf beim Lesen
I/N = 0 -> Fehler trat auf während eines Befehls
I/N = 1 -> Fehler trat auf während Eingabe-Ausgabe

Zugriff = 000 -> nicht zugeordnet
Zugriff = 001 -> User Mode Zugriff auf Daten
Zugriff = 010 -> User Mode Zugriff auf Programm
Zugriff = 011 -> nicht zugeordnet
Zugriff = 100 -> nicht zugeordnet
Zugriff = 101 -> Supervisor Mode Zugriff auf Daten
Zugriff = 110 -> Supervisor Mode Zugriff auf Programm
Zugriff = 111 -> nicht zugeordnet

Vektor	Adresse	Zuordnung
3	000C	Adressfehler

Diese Exception wird bei einem Zugriff von einem Wort oder Langwort auf einer ungeraden Speicherstelle angesprungen, z.B. "MOVE.W $1001, D2"

Normalerweise ist so ein Befehl eine Folge von einem Programmfehler. In der Exception-Behandlungsroutine geben Sie an, wie sich der Prozessor in diesem Fall zu verhalten hat.

Auch wenn Ihr Programm absichtlich solche Befehle benutzt, können Sie sie in der Exception-Behandlungs-routine implementieren.

Auf den Stack wird zusätzliche Information über den auftretenden Fehler gepushed, und zwar im gleichen Format als bei der Exception 2. Für die Beschreibung sehen Sie bitte dort nach.

Vektor	Adresse	Zuordnung
4	0010	Illegaler Befehl

Diese Exception wird erzeugt, wenn eine der folgenden Befehlsformen auftritt:

- Ein illegaler Befehl
  (Die Befehle, deren Code mit $Axxx oder $Fxxx anfangen, werden als nicht implementierte Befehle verstanden: sie werden in den Vektoren 10 und 11 abgefangen)

- ein nicht erlaubter Adressierungsmode,

- eine nicht erlaubte Befehlskombination.

Wenn wir diese Exception ABSICHTLICH erzeugen möchten, empfiehlt es sich, aus Gründen der Übersichtlichkeit und Kompatibilität den Befehl ILLEGAL ($4AFC) zu gebrauchen.

Vektor	Adresse	Zuordnung
5	0014	Division durch Null
6	0018	CHK-Befehl
7	001C	TRAPV-Befehl

Diese Vektoren sind nur vollständigkeitshalber aufgeführt. Sehen Sie bei dem entsprechenden Befehl (DIVS, DIVU, CHK, TRAPV) nach.

Vektor	Adresse	Zuordnung
8	0020	Verletzung Privilegium

Wenn der Prozessor sich im User Mode befindet und einen privilegierten Befehl ausführen soll, wird diese Exception aktiv.

Normalerweise ist das ein Programmfehler. Sie können aber auch diese Exception absichtlich betätigen.

Vektor	Adresse	Zuordnung
9	0024	Trace

Wenn das Trace-Bit im Statusregister gesetzt ist, wird nach JEDEM Befehl diese Exception gemacht. Viele Debugger nutzen diese Eigenschaft des Prozessors aus.

Vektor	Adresse	Zuordnung
10	0028	Line 1010 Emulator
11	002C	Line 1111 Emulator

Diese nicht implementierte Befehle sind speziell dafür gedacht, um den Befehlssatz softwaremäßig zu erweitern.

Befehle, deren Kode mit $A anfängt ($Axxx), erzeugen eine Exception 10. Befehle, deren Kode mit $F anfängt ($Fxxx), erzeugen eine Exception 11.

BEISPIEL 1:

Sie würden sich einen Befehl wünschen, der mit einer Instruktion einen Block im Speicher kopiert, etwa BLOCKMOVE (Quelle, Ziel, Länge). So ein Befehl bietet der 68000 Prozessor nicht. Sie können sich diesen Befehl selber erstellen, und es als Programm im Vektor 10 implementieren.

Die Abarbeitung des Befehls dauert natürlich erheblich länger, als wenn der Befehl elementar verfügbar wäre. Der Ablauf ist geringfügig schneller, als wenn der Befehl als Unterprogramm aufgerufen würde.

BEISPIEL 2:
Viele der Befehlscode des 68020 fangen mit $Fxxx an. Sie könnnen sich diese Befehle auf dem 68000 nachbauen (emulieren). Motorola hat mit diesem Verfahren Programme für den neuen Prozessor geschrieben, bevor er tatsächlich zur Verfügung stand.

Ein Vorteil der Emulation ist, daß das Programm nicht zu "wissen" braucht, daß der Befehl nicht selbständig existiert, sondern nachbgebaut wurde.

Die Vektoren 12 bis 14 sind reserviert.

Vektor	Adresse	Zuordnung
15	003C	nicht initialisiert

Der Prozessor verfügt über 256 Exceptions, jede mit einem eigenen Vektor. Jeder Vektor sollte zu der entsprechenden Exception-Behandlungsroutine zeigen. Wenn also einer der Vektoren "fehlt", würde normalerweise der Programmzähler einen unsinnigen Wert annehmen und das Programm abstürzen.

Diese Fehlerquelle wird hier abgefangen.

Wenn eine Exception auftritt und der entsprechende Vector zeigt zu einer unsinnigen Stelle, findet die hier beschriebene Exception statt. Es gibt dann für allen nicht initialisierten Exceptions eine uniforme, wohl definierte Abfertigung.

Die Vektoren 16 bis 23 sind reserviert.

Vektor	Adresse	Zuordnung
24	0060	falscher Interrupt

Wenn ein Eingabe/Ausgabe-Baustein in dem Prozessorbus einen Interrupt verursacht, aber sich bei der darauf folgenden Abfrage des Prozessors nicht meldet, sprechen wir von einem falschen (Englisch: spurious) Interrupt.

Das ist immer ein Hardwareproblem.

VORSCHLAGE ZUR FEHLERSUCHE:

- o Entfernen Sie alle Platinen, die für den Testbetrieb nicht unbedingt notwendig sind. Ist der Fehler dann verschwunden?

- o Benutzen sie vielleicht eine langsame Platine in einem schnellen Rechner?

- o Fehler in einer Leiterbahn auf der Platine, in dem Platinenstecker oder im IC-Sockel?

- o Wenn der Fehler erst nach einiger Zeit auf tritt, kann es auch mit einer zu hohen Temperatur im Gehäuse zusammenhängen (Gehäuse öffnen, Rechner einschalten).

Läßt sich der Fehler nicht auf die oben genannten Weisen einkreisen, dann brauchen Sie einen Oszillografen, um die Fehlerquelle gezielt zu suchen.

Vektor	Adresse	Zuordnung
25	0064	Ebene 1 Autovektor
26	0068	Ebene 2 Autovektor
27	006C	Ebene 3 Autovektor
28	0070	Ebene 4 Autovektor
29	0074	Ebene 5 Autovektor
30	0078	Ebene 6 Autovektor
31	007C	Ebene 7 Autovektor

Diese Vektoren werden angesprungen wenn der entsprechenden Autovector Interrupt von der Hardware ausgelöst wird. Weitere Details werden bei der Besprechung von Interrupts (Kap. 6.3) erwähnt.

Vektor	Adresse	Zuordnung
32	0080	TRAP-Befehl 0
33	0084	TRAP-Befehl 1
34	0088	TRAP-Befehl 2
..	....	....
..	....	....
46	00B8	TRAP-Befehl 14
47	00BC	TRAP-Befehl 15

Ein TRAP-Befehl erzeugt einer der obenstehenden Exceptions. Weitere Details finden Sie bei der Beschreibung des TRAP-Befehls.

Der Trap-Befehl dient typisch um z.B. Betriebssystem-Funktionen unterzubringen.

Die Vektoren 48 bis 63 sind reserviert.

Vektor	Adresse	Zuordnung
64	0100	Anwender Interrupt 0
65	0104	Anwender Interrupt 1
66	0108	Anwender Interrupt 2
..	....	....
..	....	....
254	03F8	Anwender Interrupt 190
255	03FC	Anwender Interrupt 191

Diese Vektoren werden angesprungen, wenn der entsprechenden Anwender Interrupt von der Hardware ausgelöst wird. Weitere Details werden bei der Besprechung von Interrupts (Kap. 6.3) erwähnt.

6.2.4 EXCEPTION-BEHANDLUNGSROUTINEN

In solchen Routinen legen Sie fest, welcher Vorgang im Computer eingeleitet werden soll, wenn eine Exception auftritt.

Sie können ALLE Vorgaben des Betriebssystems abändern. Wie, wird auf der nächsten Seite erwähnt.

Wie die Exceptions abgefangen werden müssen, hängt selbstverständlich davon ab, was die Aufgaben des Programms sind, und ob es nun getestet oder benutzt wird.

- Wenn Sie ein selbstgeschriebenes Programm testen, muß jede unerwünschte Exception einen Abbruch zufolge haben. Sie können dann feststellen, was im einzelnen im Programm falsch lief.

- Wenn ein Anwender Ihr Programm benutzt, muß ein Abbruch u.U. um jeden Preis vermieden werden. Es wäre besser, wenn Ihr Programm z.B. nach einem falschen Interrupt weiter rechnet, als daß es sich mit einer Fehlermeldung verabschiedet.

  Zwecks späterer Diagnose können Sie evtl. solche Fehler auf die Platte protokollieren, für den Anwender unerkennbar. Danach soll das Programm so gut möglich weiter arbeiten.

6.2.5 UMBIEGEN VON EXCEPTIONS

"Wie kann ich nun meine Exception-Behlandungsroutinen einbauen, denn sie sind doch alle bereits im Betriebssystem meines Atari festgelegt worden?" werden Sie fragen.

Eine richtige Frage. Die Antwort ist einfach. Sie heißt "Umbiegen".

In Ihrem Atari sind solche Änderungswünsche bereits vorgesehen. Deswegen zeigen alle Exception-Adressen zu einer Sprungtabelle im RAM (Schreib-Lese-Speicher).

Das bedeutet: wenn Sie diese Sprungtabelle so abändern, daß sie zu Ihrer Routine zeigt, dann wird anstelle der ursprünglichen Routine Ihre Routine angesprungen.

Sie können in Ihrer Routine, falls erwünscht, auch die ursprüngliche Exception-Behlandlungsroutine wieder aufrufen.

BEISPIEL:

Sie möchten die Tastaturbelegung so ändern, daß die Belegungen der Tasten "Y" und "Z" verwechselt werden.

Hiermit können Sie z.B. einen Atari "eindeutschen", oder eine bereits erfolgte "Eindeutschung" wieder rückgängig zu machen.

Der Vorgang ist wie hier beschrieben:

- Zuerst retten Sie in Ihrem Programm die Sprungadresse der Atari Tastaturbehandlungsroutine. Danach biegen Sie den Sprung auf "Ihre" Routine um.

- In Ihrer Routine rufen Sie die Atari Tastaturroutine als Unterprogramm auf. (Bedenken Sie bei Aufruf dieser Routine, daß sie mit einem RTE-Befehl beendet wird. Er soll in diesem Fall eine Rückkehr zu Ihrem Programm bewirken.)

- Schauen Sie in Ihrem Programm nach dem Tastaturkode. Der Kode des "Y" und "y" wird durch "Z" bzw. "z" ersetzt, der Kode des "Z" und "z" durch "Y" bzw. "y". Alle anderen Kode bleiben unverändert .

- Ihr Programm wird mit einem RTE-Befehl beendet.

Beim Laden muß das Programm **resident** gemacht, werden, d.h., daß es nicht vom nächsten Programm überschrieben wird, sondern im Speicher weiterhin vorhanden bleibt.

> Man macht ein Programm beim Laden durch entsprechende Aufrufe des Betriebssystems resident.

Das lustige ist nun, daß so ein Umbiege-Programm auch wieder "umgebogen" werden kann. Wenn Sie das Programm zweimal laden, findet die Verwechslung von "Y"und "Z" zweimal statt, so daß sich die beiden Programme in ihrer Wirkung aufheben.

Es können sich im Speicher lange Ketten von Umbiege-Programmen befinden. Insbesondere an der Uhr-Exception "hängen" manchmal lange Ketten von residenten Programmen, die im Hintergrund arbeiten.

6.3. INTERRUPTS

6.3.1 WAS IST EIN INTERRUPT ?

Am Ende jeden Befehls prüft der Prozessor die Signalpegel der drei Leitungen IPLO, IPL1 und IPL2 (IPL = Interrupt Priority Level). Der normale Ablauf ist, daß diese drei Pegel Null (+5 Volt) sind. Es passiert dann nichts besonderes: am Ende dieses Befehls wird der nächste Befehl aus dem Speicher "geholt" und ausgeführt.

Wenn aber eine von diesen drei Leitungen Eins (0 Volt) ist, dann passiert am Ende des Befehls etwas besonderes.

Diesen Vorgang nennen wir ein Interrupt.

Der Interrupt wird dadurch verursacht, daß irgendeine Schaltung im Computer etwas von dem Prozessor "will", und deswegen ein Signal auf die drei Leitungen ausgibt.

Es findet. dann eine Exception statt. Das Status Register und die Adresse des nächsten Befehls im Speicher werden auf den Stack gepushed. Die Befehlsausführung geht dann bei einer bestimmten Adresse weiter. Wir erklären später, wo.

Es findet ein Interrupt statt, wenn sich an der Hardware irgendetwas geändert hat, das den Prozessor "interessieren" könnte.

Solche Vorgängen sind z. B:

- Es wurde ein Zeichen auf der Tastatur eingetippt.
- Die Platte ist bereit, Daten entgegen zu nehmen.
- Uhr-Interrupt; dieser findet ganz regelmäßig statt.
- Sie haben die RESET-Taste gedrückt.

Es liegt jetzt an der Interruptbehandlung, ob bei einem Interrupt überhaupt etwas passiert, und wenn ja: was passiert.

6.3.2 WANN FINDET EIN INTERRUPT STATT ?

Wir haben am Anfang dieses Buches gesehen, daß das Status Wort eine Interruptmaske I0, I1, I2 hat. Diese Maske entscheidet darüber, welche Interrupts von dem Prozessor "gesehen" werden, und welche nicht.

- Hat der Interrupt mindestens den gleichen Wert als der Prozessor Priorität, findet der Interrupt statt. Dann tritt eine Exception auf. Es wird ein Sprung gemacht zu der Stelle, wo die Interrupt-Behandlungsroutine erwartet wird. Wir sagen dann: der Interrupt wird BEDIENT.
- Wenn der Interrupt eine niedrigere Ebene als der Prozessor Priorität hat, unterbleibt der Interrupt. Die Tatsache, daß eine niedrigere Interrupt Anfrage stattgefunden hat, wird aber abgespeichert. Sobald die Prozessor Priorität niedriger wird als die Interruptnummer, wird der Interrupt doch noch bedient.

Die Priorität der anliegenden Interrupts richtet sich nach dieser Tabelle:

IPL2	IPL1	IPL0	Ergebnis	Priorität
0	0	0	kein Interrupt	-
0	0	1	Interrupt 1	(niedrigste)
0	1	0	Interrupt 2	
0	1	1	Interrupt 3	
1	0	0	Interrupt 4	
1	0	1	Interrupt 5	
1	1	0	Interrupt 6	
1	1	1	Interrupt 7	(höchste)

Bild 6.2 Interrupt Prioritäten

Wenn die Interrupt Priorität größer oder gleich der Interruptmaske ist, wird der Interrupt bedient. Sonst muß der Interrupt warten. (Englisch: the interrupt is PENDING)

Es ist gefährlich, einen Interrupt zulange warten zu lassen. Kommt ein zweiter Interrupt von der gleichen Quelle bevor der erste bedient wurde, geht der erste Interrupt verloren.

- o Das bedeutet beim Empfangen von Zeichen konkret, daß Ihre Tastatur Zeichen "verschluckt", Ihre Meßdaten inkomplett sind oder Ihre Uhr zu langsam läuft.
- o Beim Senden von Zeichen ist das harmloser: es bedeutet "nur" Geschwindigkeitsverlust. Sie wird z.B. dadurch verursacht, daß Daten, die auf einer Diskette geschrieben werden müssen, eine ganze Diskettenumdrehung auf Abfertigung warten müssen, auch wenn sie nur geringfügig zu spät angeliefert werden.

Der Trick ist dann, daß die Zeichen mit hoher Priorität (= geringer Wartezeit) im Puffer geschrieben bzw. aus dem Puffer gelesen werden, und daß die weitere Abfertigung mit geringerer Priorität erfolgt. Der Programmierer wuchert hier mit Mikrosekunden!

6.3.3 WAS PASSIERT BEI EINEM INTERRUPT ?

- o Das Status Wort und der Programmzähler werden auf den Stack "gepushed", wie bei Exceptions besprochen.
- o Das Status Wort wird abgeändert:
- o Das S-Bit wird gesetzt (Supervisor Mode)
- o Das T-Bit wird zurückgesetzt (ein eventuell eingeschalteter Trace-Mode (Englisch: Trace = Spur) wird abgeschaltet).
- o Die Interruptmaske wird auf den Wert des auftretenden Interrupts gesetzt, damit zwischenzeitlich auftretende Interrupts mit niedriger Priorität warten müssen.
- o Über die Datenleitungen Ihres Computers ermittelt der Prozessor die Vektornummer der Interrupt Behandlungsroutine.

Dann wird ein Sprung zu der entsprechenden Interrupt Behandlungsroutine gemacht. Normalerweise wird diese Routine mit einem RTE-Befehl beendet, damit die Bearbeitung des Programms, das durch den Interrupt unterbrochen wurde, wieder fortgestezt werden kann.

6.3.4 WO BEFINDET SICH DIE INTERRUPTROUTINE ?

Der Interrupt wird dadurch verursacht, daß irgendeine Schaltung im Computer etwas von dem Prozessor "will". Er setzt dann auf die drei Leitungen IPLO, IPL1 und IPL2 seine Interruptnummer ab. Daran "erkennt" der Computer, welche Schaltung der Interrupt erzeugt hat.

> Das bedeutet aber nicht, daß "nur" sieben Schaltungen interruptfähig sind, denn mehrere Schaltungen können sich eine Interruptebene teilen. Das erreicht man durch eine entsprechende Verdrahtung (Daisy Chaining), die bewirkt, daß der Computer pro Interruptebene nur die jeweils aktive Schaltung "sieht".

Sobald der Prozessor einen Interrupt "gesehen" hat, fragt er den den Interruptvektor ab. Die Schaltung antwortet damit, indem sie auf die Datenleitungen der Vektornummer zeigt.

> (Tut die Schaltung dies nicht, dann haben wir einen "falschen" Interrupt, und es wird die Exception 24 erzeugt.)

Aus der Vektornummer erkennt der Prozessor, welche Position in der Tabelle 3-1 er anspringen muß, um die Interrupt Behandlungsroutine zu finden.

Aus der Vektornummer erkennt der Prozessor, welche Exception anzuspringen ist. Es gibt - um die Sache noch komplizierter zu machen - zwei Verfahren, wie sich der Prozessor die Adresse der Interrupt. Behandlungsroutine holt, und zwar den Autovektor Interrupt und den Anwender Interrupt. Ein Signal von der Schaltung zum Prozessor legt fest, ob der anlegenden Interrupt ein Autovektor Interrupt oder ein Anwender Interrupt ist.

Vektor	Adresse	Zuordnung
25	0064	Ebene 1 Autovektor Interrupt
26	0068	Ebene 2 Autovektor Interrupt
27	006C	Ebene 3 Autovektor Interrupt
28	0070	Ebene 4 Autovektor Interrupt
29	0074	Ebene 5 Autovektor Interrupt
30	0078	Ebene 6 Autovektor Interrupt
31	007C	Ebene 7 Autovektor Interrupt
64	0100	Anwender Interrupt 0
65	0104	Anwender Interrupt 1
66	0108	Anwender Interrupt 2
..	....	....
..	....	....
254	03F8	Anwender Interrupt 190
255	03FC	Anwender Interrupt 191

Bild 6.3 ZUORDNUNG VON INTERRUPTS

Anhang A
Verzeichnis der Fachbegriffe

Hier folgt eine kurze Beschreibung der Fachbegriffe, die in diesem Buch benutzt werden, Synonymen verweisen mit einem "->" zum Haupteintrag.

Für mehr Information über diese Begriffe siehe ggf. beim Stichwortverzeichnis nach.

ASCII
Abkürzung für: "American Standard Code for Information Interchange"; ist eine herstellerunabhängige Tabelle, wobei jedem Zeichen (Buchstaben, Ziffern, Lesezeichen usw.) eine eindeutige Zahl zwischen $00 und $7F zugeordnet wird.
Darüber hinaus werden zwischen $80..$FF herstellerabhängige Erweiterungen vorgenommen, die z.B. Zeichen wie ÄäÖöÜüß§ beinhalten. Die ASCII-Tabelle für Atari ist im Anhang D enthalten.

Assembler
Ein Hilfsprogramm, das ein in Assemblersprache geschriebenes Programm in ein Object-File umwandelt.

Benutzer Modus -> User Mode

Betriebssystem
Ein Programm, das den Ablauf von anderen Programmen kontrolliert und die Ansteuerung der Hardware übernimmt.

Binder -> Linker

A. Verzeichnis der Fachbegriffe

Bit
Abkürzung für Binary digIT. Ein Bit ist eine einzelne Ziffer einer Binärzahl. Sie kann einen Wert von Eins oder Null annehmen.

Byte
Eine Binärzahl von acht Bit. Die meisten Computer verarbeiten ein oder mehrere Bytes auf einmal.

Datei -> File

Debugger
Ein Programm, das dem Programmierer eine Hilfe gibt, Programme auf Fehler zu überprüfen und sie auszubessern.

Dezimal
Eine Zahl, in der jede Ziffer Werte von 0..9 annehmen kann.

Disassemblieren
Umwandlung von Maschinenkode zur Assemblersprache

Exception
Die Fähigkeit des 68000 Prozessors, den aktuellen Ablauf des Programms zu unterbrechen und etwas anderes zu tun. Exceptions werden verursacht durch Programmfehler oder durch hardware Ereignisse.

File
Eine Versammlung von Bytes, üblicherweise auf einer Platte oder einer Diskette abgespeichert. Die File (Datei) hat einen Namen: sie wird vom Betriebssystem als eine Einheit behandelt.

Halbleiterspeicher
Eine Halbleiterschaltung, die Information aufhebt, und auch wieder zurückliefert. Es gibt die folgenden Arten von Halbleiterspeichern:

- o RAM = Randorn Access Memory
  = Schreib-Lese-Speicher
- o ROM = Read Only Memory = Nur-Lese-Speicher
  Eine Art von ROMs, die EPROMs (= Erasable Programmable ROMs) kann man mit UV-Licht löschen und in einem EPROM-Programmiergerät erneut beschreiben.

Hexadezimal
Ein Zahlensystem, in dem jede Ziffer eine von 16 Werten annehmen kann (0..9, A..F, den Werten 0..15 entsprechend). Hexadezimale Zahlen lassen sich leicht in Binärzahlen um- und zurückwandeln.

IC
Abkürzung für Integrated Circuit = Integrierte Schaltung. Fast alle Elektronik Ihres Computers ist in ICs untergebracht.

Instruction Pointer -> Programmzähler

Interrupt
eine Exception, verursacht durch die Hardware

k
Eine Abkürzung für "kilo". Im Computerjargon bezieht sich k auf die Zahl 1024 = 2^{10}.

Kommentar
Im Programmtext enthaltene Zeilen oder Zeilenteile, die dem menschlichen Leser die Funktion der Programmteile erläutern sollen.

Langwort
Eine Binärzahl von 32 Bits (4 Bytes)

LIFO
Abkürzung für Last In First Out. Ein Speicherverfahren, in dem die zuletzt abgespeicherte Zahl zuerst wieder gelesen wird. Der Stack ist ein typischer LIFO-Speicher.

Linker
Ein Programm, das mehrere Object Files zu einem einzelnen ablauffähigen File zusammenfügt. In jedem der Object Files können noch externe Marken vorhanden sein, die sich auf Marken in einem anderen Object File beziehen. Der Linker setzt auch für die externen Marken den richtigen Wert ein.
(Englisch: to link = verbinden)

LSB
least significant bit = niederwertiges Bit. Das ist also bei allen Datentypen das Bit 0.

M
Eine Abkürzung für "Mega". Im ComputerJargon bezieht sich M auf die Zahl 1 048 576 = 2^{20} = 1024 x 1024.

Marke
In Assemblersprache ist eine Marke ein symbolischer Anhänger, der an Daten oder Befehle "angehängt" werden kann. Befehle können auf die Marke bezug nehmen: Der Assembler setzt dann die richtige Speicheradresse ein. (Englisch: Label)

Mnemonic
Ein Symbol, das durch den Assembler erkannt wird und das einen bestimmten Maschinenbefehl vertritt. Beispiel "JMP" ist das Mnemonic für das JUMP-Befehl.

MSB
most significant bit = hochwertiges Bit. Das ist also bei einem Byte Bit 7, bei einem Wort Bit 15 und bei einem Langwort Bit 31

Object File
Der Binärkode, wie er vom Assembler erstellt wird. Der Linker wandelt Object Files in ablauffähige Kode um.

Operand
Die Daten, die in einen Befehl verarbeitet werden sollen.

Privilegierte Befehle
Befehle, die nur dann ausgeführt werden, wenn der Prozessor in Supervisor Mode ist (Das S-Bit, Bit 13 des Status Registers ist dann gesetzt). Wenn der Prozessor im User Mode ist, werden diese Befehle nicht ausgeführt.

pop
Ein Vorgang, der den oberen Wert vom Stack entfernt.

Program Counter -> Programmzähler

Programm
Eine Gruppe von Befehlen und Daten, die eine bestimmte Funktion ausführt.

Programmzähler
Ein Register, das die Adresse des nächsten Befehls hält.

push
Ein Vorgang, der einen neuen Wert zum Stack hinzufügt.

A. Verzeichnis der Fachbegriffe

Quellkode
Der Text, den der Programmierer als Quelle für den Assembler benutzt. Der Assembler erstellt daraus den Object File.

RAM = Random Access Memory = Schreib/Lese-Speicher

Register
eine schnelle, vorläufige Speicherstelle innerhalb des Prozessors, um vorübergehend Daten abzuspeichern.

ROM = Read Only Memory = Speicher, nur zum Lesen

Schleife
Eine Reihe von Befehlen, die wiederholt durchlaufen wird. (Englisch: loop)

Speicher
Ein Teil des Computers, der Information aufhebt und auch wieder zurückliefert. Es gibt die folgenden Arten von Speichern:

- o Halbleiterspeicher (Englisch: Memory)
- o Massenspeicher: Platte, Diskette

Stack
ein Speicherverfahren im Computerspeicher, in dem das zuletzt gespeicherte Wort zuerst wieder entnommen wird. Der Stack wird häufig benutzt, um bei Aufruf von Unterprogrammen die Rückkehradresse sowie Parameter abzulegen.

Stack Frame
Ein Datenbereich, der auf den Stack reserviert wird.

Stack Pointer
Ein Register oder eine Stelle im Speicher, das die Position des letzten Wortes oder Bytes des Stacks beinhaltet.

Stapel -> Stack

Supervisor Mode
Der Modus, wenn das S-Bitr Bit 13 des Status Registers gesetzt ist. Es werden dann auch privilegierte Befehle ausgeführt.

Vektor
Eine Position im Speicher, die zu einer bestimmten Exception gehört. In dieser Speicherposition ist die Adresse der Routine enthalten, die die Exception behandeln soll.

Wort
Eine Binärzahl von 16 Bits (2 Bytes

User Mode
Der Modus, wenn das S-Bit, Bit 13 des Status Registers zurückgesetzt ist. Die privilegierten Befehle werden dann nicht ausgeführt.

Zweierkomplement
Eine Darstellung von negativen Binärzahlen, wobei die negative Zahl als das Zweierkomplement der entsprechenden positiven Zahl dargestellt wird.

A. Verzeichnis der Fachbegriffe

Addiere BCD-Zahl mit Extend-Bit
add decimal with extend **ABCD**

$Quelle_{10}$ + $Ziel_{10}$ + X -> $Ziel_{10}$

Operandgröße: ABCD. B Byte (8 Bit)
(ein Byte enthält zwei BCD-Zahlen)

Assembler Syntax:

```
ABCD.B Dn, Dn
ABCD.B -(An), -(An)   (Quelle, Ziel)
```

Beschreibung:
Der Quelloperand, das Extend-Bit und der Zieloperand werden addiert. Das Ergebnis wird im Zieloperand abgespeichert. Die Addition findet als BCD-Arithmetik statt.

Die Operanden können in zwei Arten adressiert werden:

Dn Datenregister zu Datenregister. Die Operanden sind die niederwertigsten Bytes des Datenregisters.

-(An) Von Speicherplatz zu Speicherplatz. Diese Art ist gedacht, um mehrere BCD-Zahlen im Speicher zu addieren. Die Operanden werden durch das Adressregister im Prä-dekrement-Mode adressiert.

Da der 68000 BCD-Zahlen mit dem niederwertigsten Byte auf der höchsten Speicherstelle ablegt, können Sie auf der höchsten Adresse anfangen, um mehrere Bytes automatisch abzuarbeiten.

Für eine weitere Beschreibung siehe bei NBCD.

Für eine Darstellung von BCD-Ziffern siehe Kap. 3.6

ABCD

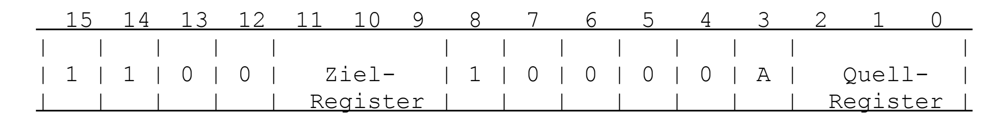
Addiere BCD-Zahl mit Extend-Bit
add decimal with extend

X	N	Z	V	C
*	?	*	?	*

Condition Code Register:

C wird gesetzt, wenn ein (dezimaler) Übertrag generiert wird. Wird sonst zurückgesetzt.
V nicht definiert
Z wird gelöscht, wenn das Ergebnis ungleich Null ist. Bleibt sonst unverändert.
N nicht definiert
X Erhält, den gleichen Wert wie das C-Bit.

Format des Befehlswortes:



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	Ziel-Register			1	0	0	0	0	A	Quell-Register		

Bit 11..9 Registerfeld: wählt eines der acht Daten- oder Adressregistern als Zieloperand an.

Bit 3 Das A-bit wählt die Adressierungsart an.
0 Adressierungsart Dn: Die Operation erfolgt von Datenregister zu Datenregister.
1 Adressierungsart -(An): Die Operation erfolgt von Speicherplatz zu Speicherplatz. Die Operanden werden durch das Adressregister in Prä-dekrement-mode adressiert. Siehe Kap. 5.

Bit 2..0 Registerfeld: wählt einer der acht Datenoder Adressregistern als Quelloperand an.

Siehe auch: ADDX, NBCD, SBCD

Addiere binär
add

ADD

Quelle + Ziel -> Ziel

Operandgröße: ADD.B Byte (8 Bit)
ADD.W Wort (16 Bit)
ADD.L Langwort (32 Bit)

Assembler Syntax:	**Operation:**
ADD.x <ea>, Dn	<ea> + Dn -> Dn
ADD.x Dn, <ea>	Dn + <ea> -> <ea>
(x entspricht B, W, L)	

Beschreibung:

Der Quelloperand wird binär zum Zieloperand addiert, und das Ergebnis wird im Zieloperand abgespeichert.

Einer der beiden Operanden muß ein Datenregister sein.

Die Größe des Operanden, sowie die Angabe, welcher Operand das Datenregister ist, sind im Mode-Feld enthalten.

ADD

Addiere binär
add

```
   X    N    Z    V    C
 ________________________
|    |    |    |    |    |
| *  | *  | *  | *  | *  |
|____|____|____|____|____|
```

Condition Code Register:

C wird gesetzt, wenn ein Übertrag generiert wird. Wird sonst zurückgesetzt.
V wird gesetzt, wenn ein Überlauf generiert wird. Wird sonst zurückgesetzt.
Z wird gesetzt, wenn das Ergebnis gleich Null ist. Wird sonst zurückgesetzt.
N wird gesetzt, wenn das Ergebnis negativ ist. Wird sonst zurückgesetzt.
X Erhält den gleichen Wert als das C-Bit.

Format des Befehlswortes:

```
| 15  14  13  12| 11  10  9   8 | 7   6   5   4 | 3   2   1   0 |
|   |   |   |   |            |            |                     |
| 1 | 1 | 0 | 1 | Register   |Operations-|    Effektive Adresse |
|___|___|___|___|____Dn______|___Mode____|___Mode_____|_Register_|
```

Bit 11..9 Registerfeld: wählt eines der acht Datenregister an.

Bit 8..6 Feld Operationsmode:

ADD.B	ADD.W	ADD.L	Operation
000	001	010	<ea> + Dn -> Dn
100	101	110	Dn + <ea> -> <ea>

Addiere binär
add

ADD

Bit 5..0 Wenn die Effektive Adresse der Quelloperand ist (also <ea> + Dn -> Dn), sind die folgenden Adressierungsarten erlaubt:

Adr.-Art	Mode	Reg
Dn	000	R:Dn
An *)	001	R:An
(An)	010	R:An
(An)+	011	R:An
-(An)	100	R:An
d16 (An)	101	R:An
d8(An,Xi)	110	R:An

Adr.-Art	Mode	Reg
xxx.W	111	000
xxx.L	111	001
d16(PC)	111	010
d8(PC,Xi)	111	011
#<data>	111	100
Erläuterung siehe Kapitel 5		

*) Adressierungsart An nicht für Byte-Befehle erlaubt

Bit 5..0 Wenn die Effektive Adresse der Zieloperand ist (also Dn + <ea> -> <ea>), sind die folgenden Adressierungsarten erlaubt:

Adr.-Art	Mode	Reg
Dn	nicht erlaubt	
An	nicht erlaubt	
(An)	010	R:An
(An)+	011	R:An
-(An)	100	R:An
d16 (An)	101	R:An
d8(An,Xi)	110	R:An

Adr.-Art	Mode	Reg
xxx.W	111	000
xxx.L	111	001
d16(PC)	nicht erlaubt	
d8(PC,Xi)	nicht erlaubt	
#<data>	nicht erlaubt	
Erläuterung siehe Kapitel 5		

HINWEIS:
Wenn der Zieloperand ein Datenregister sein soll, kann er nicht mit der Adressierungsart <ea> angewählt werden, sondern nur mit der Adressierungsart Dn.

ADD

Addiere binär
add

Benutzen Sie

ADDA:	wenn der Zieloperand ein Adressregister ist;
ADDI oder ADDQ:	wenn einer der Operanden eine Konstante ist.
ADDX:	wenn ein anderes Verhalten des Z-Bits erwünscht ist.

Siehe auch: SUB

Addiere Adresse
add address

ADDA

Quelle + ziel -> ziel

Operandgröße: ADDA.W Wort (16 Bit)
ADDA.L Langwort (32 Bit)

Assembler Syntax: ADDA.x <ea>, An (Quelle, Ziel)
(x entspricht W, L)

Beschreibung:

Der Quelloperand in <ea> wird binär zum Ziel-Adressregister An addiert. Das Ergebnis wird im Ziel-Adressregister An abgespeichert.
Vom Zielregister An werden sämtliche Bytes angewendet, unabhängig von der Operandgröße.

X	N	Z	V	C
-	-	-	-	-

Condition Code Register:
keine Änderungen

ADDA

Addiere Adresse
add address

Format des Befehlswortes:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	Register Dn			Operations-Mode			Effektive Adresse Mode			Effektive Adresse Register		

Bit 11..9 Registerfeld: wählt eines der acht Adressregister An an.
Es ist der Zieloperand.

Bit 8..6 Feld Operations-Mode:

011 ADDA.W - Wort-Befehl. Der Quellopperand wird mit dem gleichen Vorzeichen auf 32 Bit erweitert, und vom Ziel-Adressregister werden sämtliche 32 Bits angewendet.

111 ADDA.L - Langwort-Befehl

Bit 5..0 Die Effektive Adresse wählt den Quell-operand an. Alle Adressierungsarten sind erlaubt:

Adr.-Art	Mode	Reg
Dn	000	R:Dn
An	001	R:An
(An)	010	R:An
(An)+	011	R:An
-(An)	100	R:An
d16(An)	101	R:An
d8(An,Xi)	110	R:An

Adr.-Art	Mode	Reg
xxx.W	111	000
xxx.L	111	001
d16(PC)	111	010
d8(PC,Xi)	111	011
#<data>	111	100
Erläuterung siehe Kapitel 5		

Siehe auch: ABCD, ADD, ADDI, ADDQ, ADDX, SUBA

Addiere Konstante
add immediate

ADDI

Konstante + Ziel -> Ziel

Operandgröße: ADDI.B Byte (8 Bit)
ADDI.W Wort (16 Bit)
ADDI.L Langwort (32 Bit)

Assembler Syntax: ADDI.x #<data>, <ea> (Quelle, Ziel)
(x entspricht B, W, L)

Beschreibung:

Die Konstante, die im Speicher unmittelbar dem Befehlswort folgt, wird binär zum Zieloperand <ea> addiert. Das Ergebnis wird im Zieloperand <ea> abgespeichert.

Die Größe der Konstante entspricht der Operandgröße.

Condition Code Register:

X	N	Z	V	C
*	*	*	*	*

C wird gesetzt, wenn ein Übertrag generiert wird. Wird sonst zurückgesetzt.
V wird gesetzt, wenn ein Überlauf generiert wird. Wird sonst zurückgesetzt.
Z wird gesetzt, wenn das Ergebnis gleich Null ist. Wird sonst zurückgesetzt.
N wird gesetzt, wenn das Ergebnis negativ ist. Wird sonst zurückgesetzt.
X Erhält den gleichen Wert wie das C-Bit.

ADDI

Addiere Konstante
add immediate

Format des Befehlswortes:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	Größe		Effektive Adresse Mode			Register		
1. Argumentwort: Wort Daten								bzw. Byte Daten							
2. Argumentwort: Langwort Daten (einschließlich voriges Wort)															

Bit 7..6 Größe-Feld:
00 Byte-Befehl ADDI.B
01 Wort-Befehl AUDI.W
10 Langwort-Befehl ADDI.L

Aufbau der Argumentwörter siehe Kap. 3.8

Bit 5..0 Die Effektive Adresse wählt den Zieloperand an. Die folgenden Adressierungsarten sind erlaubt:

Adr.-Art	Mode	Reg
Dn	000	R:Dn
An	nicht erlaubt	
(An)	010	R:An
(An)+	011	R:An
-(An)	100	R:An
d16(An)	101	R:An
d8(An,Xi)	110	R:An

Adr.-Art	Mode	Reg
xxx.W	111	000
xxx.L	111	001
d16(PC)	nicht erlaubt	
d8(PC,Xi)	nicht erlaubt	
#<data>	nicht erlaubt	
Erläuterung siehe Kapitel 5		

Argumentwort:
Bit 7..6 = 00 -> Datenfeld ist die niederwertige Hälfte des 1. Argumentwortes
Bit 7..6 = 01 -> Datenfeld ist das 1. Argumentwort
Bit 7..6 = 10 -> Datenfeld ist 1. + 2. Argumentwort

Siehe auch: ABCD, ADD, ADDA, ADDQ, ADDX, SUBI

Addiere Konstante "quick" (1..8)
add quick

ADDQ

Konstante + Ziel -> Ziel

Operandgröße: ADDQ.B Byte (8 Bit)
ADDQ.W Wort (16 Bit)
ADDQ.L Langwort (32 Bit)

Assembler Syntax: ADDQ.x #<data>, <ea> (Quelle, Ziel)
(x entspricht B, W, L)

Beschreibung:
Dieser Befehl addiert die Konstante zum Zieloperand <ea>. Das Ergebnis wird im Zieloperand <ea> abgespeichert.
Die Konstante muß zwischen l und 8 liegen.

Condition Code Register:

X	N	Z	V	C
*	*	*	*	*

c wird gesetzt, wenn ein Übertrag generiert wird.
Wird sonst zurückgesetzt.
v wird gesetzt, wenn ein Überlauf generiert, wird.
Wird sonst zurückgesetzt.
z wird gesetzt, wenn das Ergebnis gleich Null ist.
Wird sonst zurückgesetzt.
N wird gesetzt, wenn das Ergebnis negativ ist.
Wird sonst zurückgesetzt.
X Erhält den gleichen Wert wie das C-Bit.

HINWEIS: Wenn der Zieloperand ein Adressregister ist (Adressierungsart An), wird das Condition Code Register nicht geändert.

ADDQ

Addiere Konstante "quick" (1..8)
add quick

Format des Befehlswortes:

15	14	13	12	11 10 9	8	7 6	5 4 3	2 1 0
0	1	0	1	D a t e n	0	Größe	Effektive Adresse Mode	Effektive Adresse Register

Bit 11..9 Datenfeld:
000 entspricht Konstante 1, 001 entspricht
Konstante 2 usw. bis 111 Konstante 8 entspricht.

Bit 7..6 Größe-Feld:
00 Byte-Befehl ADDQ.B
01 Wort-Befehl ADDQ.W
10 Langwort-Befehl ADDQ.L

Bit 5..0 Die Effektive Adresse wählt den Zieloperand an. Die folgenden Adressierungsarten sind erlaubt:

Adr.-Art	Mode	Reg
Dn	000	R:Dn
An *)	001	R:An
(An)	010	R:An
(An)+	011	R:An
-(An)	100	R:An
d16(An)	101	R:An
d8(An,Xi)	110	R:An

Adr.-Art	Mode	Reg
xxx.W	111	000
xxx.L	111	001
d16(PC)	nicht erlaubt	
d8(PC,Xi)	nicht erlaubt	
#<data>	nicht erlaubt	
Erläuterung siehe Kapitel 5		

*)Adressierungsart An nicht für Byte-Befehle erlaubt

Benutzen Sie
ADDI: wenn die Konstante außerhalb des Bereiches 1..8 liegt.

Siehe auch: ABCD, ADD, ADDA, ADDX, SUBQ

Addiere mit Extend-Bit
add extended **ADDX**

Quelle + Ziel + X -> Ziel

Operandgröße: ADDX.B Byte (8 Bit)
ADDX.W Wort (16 Bit)
ADDX.L Langwort (32 Bit)

Assembler Syntax: ADDX.x Dn, Dn
ADDX.x -(An), -(An) (Quelle,Ziel)
(x entspricht B, W, L)

Beschreibung:

Der Quelloperand, das Extend-Bit und der Zieloperand werden addiert. Das Ergebnis wird im Zieloperand abgespeichert.

Die Operanden können in zwei Arten adressiert werden:

Dn Datenregister zu Datenregister. Die Operanden sind die niederwertigsten Bytes der Datenregister.

-(An) Von Speicherplatz zu Speicherplatz. Diese Art ist gedacht, um mehrere Binärzahlen im Speicher zu addieren. Die Operanden werden durch das Adressregister in Prä-dekrement-Mode adressiert.

Da der 68000 Daten mit dem niederwertigsten Byte auf der höchsten Speicherstelle ablegt, können Sie auf der höchsten Adresse anfangen, um mehrere Bytes automatisch abzuarbeiten.

Für eine weitere Beschreibung siehe bei NEGX.

ADDX

Addiere mit Extend-Bit
add extended

Condition Code Register:

X	N	Z	V	C
*	*	*	*	*

C wird gesetzt, wenn ein Übertrag generiert wird.
Wird sonst zurückgesetzt.
V wird gesetzt, wenn ein Überlauf generiert wird.
Wird sonst zurückgesetzt.
Z wird gelöscht, wenn das Ergebnis ungleich Null ist.
Bleibt sonst unverändert.
N wird gesetzt, wenn das Ergebnis negativ ist.
Wird sonst zurückgesetzt.
X Erhält den gleichen Wert wie das C-Bit.

Addiere mit Extend-Bit
add extended

ADDX

Format des Befehlswortes:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	Ziel-Register			1	0	0	0	0	A	Quell-Register		

Bit 11..9 Registerfeld: wählt eines der acht Daten- oder Adressregister als Zieloperand an.

Bit 3 Das A-Bit wählt die Adressierungsart an.

0 Adressierungsart Dn: Die Operation erfolgt von Datenregister zu Datenregister

1 Adressierungsart -(An): Die Operation erfolgt von Speicherplatz zu Speicherplatz. Die Operanden werden durch das Adressregister in Prädekrement-mode adressiert. Siehe Kap. 5.

Bit 2..0 Registerfeld: wählt eines der acht Daten- oder Adressregister als Quelloperand an.

Siehe auch: ADDQ, ADDI, ABCD, ADD, ADDA, SUBX

AND

Logisches UND
and logical

Quelle ^ Ziel -> Ziel

Operandgröße: AND.B Byte (8 Bit)
AND.W Wort (16 Bit)
AND.L Langwort (32 Bit)

Assembler Syntax:	**Operation:**
AND.x <ea>, Dn	<ea> ^ Dn -> Dn
AND.x Dn, <ea>	Dn ^ <ea> -> <ea>
(x entspricht B, W, L)	

Beschreibung:

Der Quelloperand wird mit dem Zieloperand bitweise UND-verknüpft und das Ergebnis wird im Zieloperand abgespeichert.

Zur Erinnerung die UND-Verknüpfungen:

0 ^ 0 = 0
0 ^ 1 = 0
1 ^ 0 = 0
1 ^ 1 = 1

Das Ergebnisbit wird gesetzt, wenn das eine und das andere Eingangsbit gesetzt sind.

Einer der beiden Operanden muß ein Datenregister sein.

Die Größe des Operanden sowie die Angabe, welcher Operand das Datenregister ist, sind im Mode-Feld enthalten.

Logisches UND
and logical

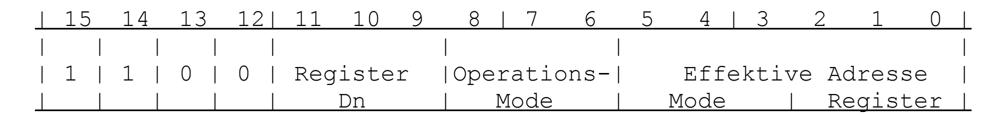
AND

Condition Code Register:

X	N	Z	V	C
-	*	*	0	0

C wird zurückgesetzt.
V wird zurückgesetzt.
Z wird gesetzt, wenn das Ergebnis gleich Null ist.
  Wird sonst zurückgesetzt.
N wird gesetzt, wenn das hochwertige Bit des
  Ergebnisses gesetzt ist (zeigt ein negatives
  Ergebnis).
  Wird sonst zurückgesetzt.
X bleibt unverändert.

Format des Befehlswortes:



15	14	13	12	11 10 9	8 7 6	5 4 3	2 1 0
1	1	0	0	Register Dn	Operations-Mode	Effektive Adresse Mode	Effektive Adresse Register

Bit 11..9 Registerfeld: wählt eines der acht Datenregister Dn an.

Bit 8..6 Feld Operationsmode:

AND.B	AND.W	AND.L	Operation
000	001	010	<ea> ^ Dn -> Dn
100	101	110	Dn ^ <ea> -> <ea>

AND

Logisches UND
and logical

Bit 5..0 Wenn die Effektive Adresse der **Quelloperand** ist (also <ea> ^ Dn -> Dn), sind die folgenden Adressierungsarten erlaubt:

Adr.-Art	Mode	Reg
Dn	000	R:Dn
An	nicht erlaubt	
(An)	010	R:An
(An)+	011	R:An
-(An)	100	R:An
d16(An)	101	R:An
d8(An,Xi)	110	R:An

Adr.-Art	Mode	Reg
xxx.W	111	000
xxx.L	111	001
d16(PC)	111	010
d8(PC,Xi)	111	011
#<data>	111	100
Erläuterung siehe Kapitel 5		

Bit 5..0 Wenn die Effektive Adresse der **Zieloperand** ist (also Dn ^ <ea> -> <ea>), sind die folgenden Adressierungsarten erlaubt:

Adr.-Art	Mode	Reg
Dn	nicht erlaubt	
An	nicht erlaubt	
(An)	010	R:An
(An)+	011	R:An
-(An)	100	R:An
d16(An)	101	R:An
d8(An,Xi)	110	R:An

Adr.-Art	Mode	Reg
xxx.W	111	000
xxx.L	111	001
d16(PC)	nicht erlaubt	
d8(PC,Xi)	nicht erlaubt	
#<data>	nicht erlaubt	
Erläuterung siehe Kapitel 5		

HINWEIS:
Wenn der Zieloperand ein Datenregister sein soll, kann er nicht mit der Adressierungsart <ea> angewählt werden, sondern nur mit der Adressierungsart Dn.

Logisches UND
and logical

AND

Benutzen Sie

ANDI:	wenn einer der Operanden eine Konstante ist.

Siehe auch: OR, EOR, NOT, TST

ANDI

UND mit Konstante
and immediate

Konstante ^ Ziel -> Ziel

Operandgröße: ANDI.B Byte (8 Bit)
ANDI.W Wort (16 Bit)
ANDI.L Langwort (32 Bit)

Assembler Syntax: ANDI.x #<data>, <ea> (Quelle, Ziel)
(x entspricht B, W, L)

Beschreibung:

Die Konstante, die im Speicher unmittelbar dem Befehlswort folgt, wird mit dem Zieloperand <ea> bitweise UND-verknüpft. Das Ergebnis wird im Zieloperand <ea> abgespeichert.

Die Größe der Konstante entspricht der Operandgröße.

Condition Code Register:

X	N	Z	V	C
-	*	*	0	0

C wird zurückgesetzt.
V wird zurückgesetzt.
Z wird gesetzt, wenn das Ergebnis gleich Null ist, Wird sonst zurückgesetzt.
N wird gesetzt, wenn das hochwertige Bit des Ergebnisses gesetzt ist (zeigt ein negatives Ergebnis). Wird sonst zurückgesetzt.
X bleibt unverändert.

UND mit Konstante
and immediate

ANDI

Format des Befehlswortes:

```
| 15  14  13  12| 11  10   9   8 | 7   6   5   4 | 3   2   1   0 |
|   |   |   |   |   |   |   |   |       |                       |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | Größe |   Effektive Adresse   |
|   |   |   |   |   |   |   |   |       |  Mode     |  Register |
|                                |                              |
| 1. Argumentwort: Wort Daten    |   bzw.  Byte Daten           |
|                                |                              |
|                                                               |
| 2. Argumentwort: Langwort Daten (einschließlich voriges Wort) |
|                                                               |
```

Bit 7..6 Größe-Feld:
00 Byte-Befehl ANDI.B
01 Wort-Befehl ANDI.W
10 Langwort-Befehl ANDI.L

Aufbau der Argumentwörter siehe Kap. 3.8

Bit 5..0 Die Effektive Adresse wählt den Zieloperand an. Die folgenden Adressierungsarten sind erlaubt:

Adr.-Art	Mode	Reg
Dn	000	R:Dn
An	nicht erlaubt	
(An)	010	R:An
(An)+	011	R:An
-(An)	100	R:An
d16(An)	101	R:An
d8(An,Xi)	110	R:An

Adr.-Art	Mode	Reg
xxx.W	111	000
xxx.L	111	001
d16(PC)	nicht erlaubt	
d8(PC,Xi)	nicht erlaubt	
#<data>	nicht erlaubt	
Erläuterung siehe Kapitel 5		

Argumentwort:
Bit 7..6 = 00 -> Datenfeld ist die niederwertige Hälfte des 1. Argumentwortes
Bit 7..6 = 01 -> Datenfeld ist das 1. Argumentwort
Bit 7..6 = 10 -> Datenfeld ist 1. +2. Argumentwort

Siehe auch: AND, ORI, EORI, NOT, TST

ANDI zum CCR

UND mit Konstante zum CCR
and immediate to condition codes

Konstante ^ CCR -> CCR

Operandgröße: Byte (8 Bit)

Assembler Syntax: ANDI #<data>, CCR (Quelle, Ziel)

Beschreibung:

Die Konstante, die im Speicher unmittelbar dem Befehlswort folgt, wird mit dem Condition Code Register bitweise UND-verknüpft. Das Ergebnis wird im Condition Code Register abgespeichert.

Die Operandgröße ist ein Byte.

Condition Code Register:

X	N	Z	V	C
*	*	*	*	*

C wird zurückgesetzt, wenn Bit 0 der Konstante zurückgesetzt ist. Bleibt sonst unverändert.
V wird zurückgesetzt, wenn Bit 1 der Konstante zurückgesetzt ist. Bleibt sonst unverändert.
Z wird zurückgesetzt, wenn Bit 2 der Konstante zurückgesetzt ist. Bleibt sonst unverändert.
N wird zurückgesetzt, wenn Bit 3 der Konstante zurückgesetzt ist. Bleibt sonst unverändert.
X wird zurückgesetzt, wenn Bit 4 der Konstante zurückgesetzt ist. Bleibt sonst unverändert.

UND mit Konstante zum CCR
and immediate to condition codes

ANDI zum CCR

Format des Befehlswortes:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	0	0	0	1	1	1	1	0	0
0	0	0	0	0	0	1	0	Konstante (8 Bit)							

Benutzen Sie

ANDI zum SR:	wenn Sie auch das System Byte UND-verknüpfen möchten.
ORI zum CCR:	wenn Sie das CCR ODER-verknüpfen möchten.
EORI zum CCR:	wenn Sie das CCR Exclusiv-ODER-verknüpfen möchten.
MOVE zum CCR:	wenn Sie das CCR ohne Rücksicht auf die bestehenden Bits ändern möchten.

ANDI zum SR

UND mit Konstante zum SR
and immediate to status register
(privilegierter Befehl)

Wenn Supervisor Mode: Konstante ^ SR -> SR

Wenn User Mode: Auslösung Exception 8 (Kap 6)
(Verletzung Privilegium)

Operandgröße: Wort (16 Bit)

Assembler Syntax: ANDI #<data>, SR (Quelle, Ziel)

Beschreibung:

Wenn der Prozessor sich im Supervisor Mode befindet, wird die Konstante, die im Speicher unmittelbar dem Befehlswort folgt, mit dem Status Register bitweise UND-verknüpft. Das Ergebnis wird im Status Register abgespeichert.

Wenn der Prozessor dagegen im User Mode ist, wird eine Exception ausgelöst.

Das Status Register wird in Kap. 2 beschrieben. Die Operandgröße ist ein Wort (16 Bit).

UND mit Konstante zum SR
and immediate to status register
(privilegierter Befehl)

ANDI zum SR

Condition Code Register:

X	N	Z	V	C
*	*	*	*	*

C wird zurückgesetzt, wenn Bit 0 der Konstante zurückgesetzt ist. Bleibt sonst unverändert.
V wird zurückgesetzt, wenn Bit 1 der Konstante zurückgesetzt ist. Bleibt sonst unverändert.
Z wird zurückgesetzt, wenn Bit 2 der Konstante zurückgesetzt ist. Bleibt sonst unverändert.
N wird zurückgesetzt, wenn Bit 3 der Konstante zurückgesetzt ist. Bleibt sonst unverändert.
X wird zurückgesetzt, wenn Bit 4 der Konstante zurückgesetzt ist. Bleibt sonst unverändert.

Format des Befehlswortes:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	0	0	1	1	1	1	1	0	0
Konstante (16 Bit)															

Benutzen Sie

ORI zum SR:	wenn Sie das SR ODER-verknüpfen möchten.
EORI zum SR:	wenn Sie das SR Exclusiv-ODER-verknüpfen möchten.
MOVE zum SR:	wenn Sie das SR ohne Rücksicht auf die bestehenden Bits ändern möchten.
ANDI zum CCR:	wenn Sie nur das Condition Code Register ändern möchten.

ASL

Arithmetisches Schieben nach links
arithmetic shift left

Ziel verschoben durch <Zahl> -> Ziel

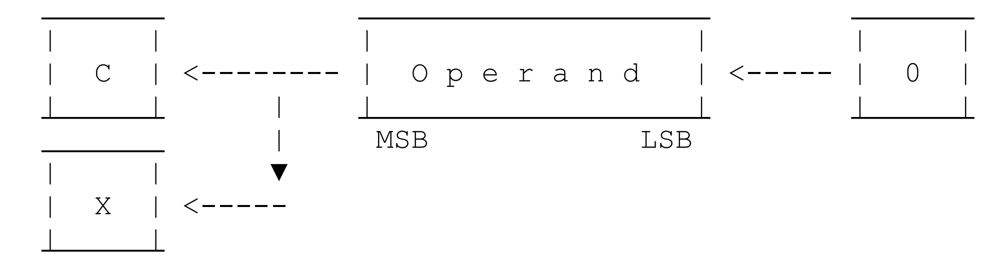
```
Operandgröße:  ASL.B  Byte (8 Bit)
               ASL.W  Wort (16 Bit)
               ASL.L  Langwort (32 Bit)
               ASL  Wort (16 Bit)

Assembler      ASL.x  Dn, Dn  (Quelle, Ziel)
Syntax:        ASL.x  #<data>, Dn
                  (x entspricht B, W, L)
               ASL    <ea>
```

Beschreibung:
Die Bits des Operanden werden nach links verschoben. Bei jedem Schiebeschritt passiert folgendes:

- Das C-Bit und das X-Bit erhalten den Wert des hochwertigen Bits.
- Danach erhält das hochwertige Bit den Wert des Bits rechts daneben. Dieses Bit erhält dann den Wert seines rechten Nachbarns usw, bis Bit 1 den Wert von Bit 0 erhält.
- Zum Schluß erhält Bit 0 den Wert Null.

Für die Gesamtzahl der Schiebeschritte siehe weiter unten.



Arithmetisches Schieben nach links
arithmetic shift left

ASL

Ein ASL um n Positionen bedeutet, daß der Operand - als Binärzahl mit Vorzeichen aufgefaßt - mit 2^n multipliziert wird. Einen eventuellen Überlauf kann man am V-Bit feststellen.

HINWEIS:
Der Befehl ist weitgehend ähnlich zum Befehl LSL.
Der Unterschied ist der, daß das LSL das V-Bit zurücksetzt.

Es gibt drei Befehlsformen.

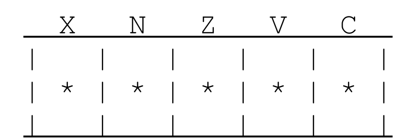
- Der Befehl
  ASL.x #<data>, Dn (x entspricht B, W, L)
  schiebt ein Datenregister nach links um soviele Positionen, wie in der Konstante angegeben ist. Die Maximalzahl ist acht.

- Der Befehl
  ASL.x Dnr Dn (x entspricht B, W, L)
  schiebt ein Datenregister nach links. Ein zweites Datenregister legt fest, um wieviele Positionen geschoben wird.

- Der Befehl
  ASL <ea>
  schiebt eine Speicherstelle (16 Bit) um eine Position nach links.

ASL

Arithmetisches Schieben nach links
arithmetic shift left



X	N	Z	V	C
*	*	*	*	*

Condition Code Register:

C erhält den Wert, der zuletzt aus dem hochwertigen Bit des Operanden herausgeschoben wurde. Wird zurückgesetzt bei Schieben um Null Positionen.

V wird gesetzt, wenn das hochwertige Bit des Operanden sich während des Verschiebens mindestens einmal ändert. Wird sonst zurückgesetzt.

Z wird gesetzt, wenn das Ergebnis gleich Null ist. Wird sonst zurückgesetzt.

N wird gesetzt, wenn das hochwertige Bit des Ergebnisses gesetzt ist. Wird sonst zurückgesetzt.

X erhält den Wert, der zuletzt aus dem hochwertigen Bit des Operanden herausgeschoben wurde. Bleibt unverändert bei Schieben um Null Positionen.

Arithmetisches Schieben nach links
arithmetic shift left

ASL

Assembler Syntax: **ASL.x Dn, Dn**
ASL.x #<data>, Dn
(x entspricht B, W, L)

Dazu gehört das folgende Format des Befehlswortes:

15	14	13	12	11 10 9	8	7 6	5	4	3	2 1 0
1	1	1	0	Zähl-Register	1	Größe	i	0	0	Daten Register

Bit 11..9 Zählregister-Feld
Wenn i = 0:
Die Bits 11.. 9 geben an, um wieviele Positionen die Bits des Zieloperanden nach links verschoben werden. Dabei entspricht 001 einer Position usw., bis 111 sieben Positionen entspricht. 000 entspricht aber acht Positionen. (Konstante)
Wenn i = 1:
Die Bits 11.. 9 wählen ein Datenregister Dn an. Die niederwertigen 6 Bits des Datenregisters Dn geben an, um wieviele Positionen die Bits des Zieloperanden nach links verschoben werden.

Bit 7..6 Größe-Feld
00 Byte-Befehl ASL.B
01 Wort-Befehl ASL.W
10 Langwort-Befehl ASL.W

Bit 5 i-Feld
0 Die Bits 11..9 beziehen sich auf eine Konstante
1 Die Bits 11..9 beziehen sich auf ein Datenregister.

Bit 2..0 Register-Feld
wählt ein Datenregister als Zieloperand an.

ASL

Arithmetisches Schieben nach links
arithmetic shift left

Assembler Syntax: ASL <ea>

Die 16 Bits des angewählten Speicherwortes werden um eine Position nach links verschoben.

Dazu gehört das folgende Format des Befehlswortes:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	0	0	1	1	1	Effektive Adresse					
										Mode			Register		

Bit 5..0 Wählt die Effektive Adresse des Quelloperanden an. Die folgenden Adressierungsarten sind erlaubt:

Adr.-Art	Mode	Reg
Dn	nicht erlaubt	
An	nicht erlaubt	
(An)	010	R:An
(An)+	011	R:An
-(An)	100	R:An
d16(An)	101	R:An
d8(An,Xi)	110	R:An

Adr.-Art	Mode	Reg
xxx.W	111	000
xxx.L	111	001
d16(PC)	nicht erlaubt	
d8(PC,Xi)	nicht erlaubt	
#<data>	nicht erlaubt	
Erläuterung siehe Kapitel 5		

Für die Adressierungsart Dn siehe auf der vorherigen Seite.

Benutzen Sie
ASR: wenn nach rechts geschoben werden soll
LSL: wenn das V-Bit zurückgesetzt werden soll.

Siehe auch: LSR, ROL, ROR, ROXL, ROXR, SWAP

Arithmetisches Schieben nach rechts
arithmetic shift right

ASR

Ziel verschoben durch <Zahl> -> Ziel

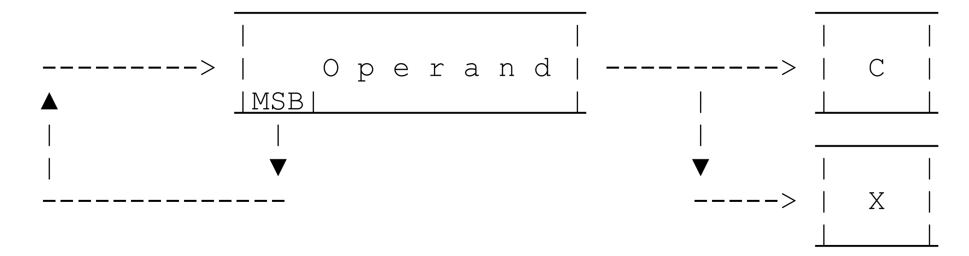
Operandgröße: ASR.B Byte (8 Bit)
ASR.W Wort (16 Bit)
ASR.L Langwort (32 Bit)
ASR Wort (16 Bit)

Assembler Syntax:
```
ASR.x  Dn, Dn  (Quelle, Ziel)
ASR.x  #<data>, Dn
   (x entspricht B, W, L)
ASR   <ea>
```

Beschreibung:
Die Bits des Operanden werden nach rechts verschoben. Bei jedem Schiebeschritt passiert folgendes:

- o Bit 0 gibt seinen Wert an das C-Bit und das X-Bit ab.
- o Danach gibt Bit 1 seinen Wert an Bit 0, und Bit 2 seinen Wert an Bit 1 usw., bis das hochwertige Bit seinen Wert an seinen rechten Nachbarn abgibt.
- o Das hochwertige Bit behält aber während des ganzen Schiebvorganges seinen eigenen Wert.

Für die Gesamtzahl der Schiebeschritte siehe weiter unten.



ASR

Arithmetisches Schieben nach rechts
arithmetic shift right

Ein ASR um n Positionen bedeutet, daß der Operand - als Binärzahl mit Vorzeichen aufgefaßt - durch 2^n dividiert wird: der Operand erhält den Wert des Quotienten. Der Wert des Restes ist so nicht feststellbar, er läßt sich mit DIVU ermitteln. Das Vorzeichen verbleibt unverändert in Bit 0.

HINWEIS:
Der Befehl ist weitgehend ähnlich zum Befehl LSR.
Der Unterschied ist der, daß das LSR das hochwertige Bit des Operanden zurücksetzt.

Es gibt drei Befehlsformen.

- Der Befehl
  ASR.x i<data>, Dn (x entspricht B, W, L)
  schiebt ein Datenregister nach rechts um soviele Positionen, wie in der Konstante angegeben ist. Die Maximalzahl ist acht.

- Der Befehl
  ASR.x Dn, Dn (x entspricht B, W, L)
  schiebt ein Datenregister nach rechts. Ein zweites Datenregister legt fest, um wieviele Positionen geschoben wird.

- Der Befehl
  ASR <ea>
  schiebt eine Speicherstelle (16 Bit) um eine Position nach rechts.

Arithmetisches Schieben nach rechts
arithmetic shift right

ASR

X	N	Z	V	C
*	*	*	0	*

Condition Code Register:

C erhält den Wert, der zuletzt aus dem niederwertigen Bit des Operanden herausgeschoben wurde. Wird zurückgesetzt bei Schieben um Null Positionen.
V wird zurückgesetzt.
Z wird gesetzt, wenn das Ergebnis gleich Null ist. Wird sonst zurückgesetzt.
N wird gesetzt, wenn das hochwertige Bit des Ergebnisses gesetzt ist (zeigt ein negatives Ergebnis).
Wird sonst zurückgesetzt.
X erhält den Wert, der zuletzt aus dem niederwertigen Bit des Operanden herausgeschoben wurde. Bleibt unverändert bei Schieben um Null Positionen.

ASR

Arithmetisches Schieben nach rechts
arithmetic shift right

Assembler Syntax: ASR.x Dn , Dn
ASR.x #<data>, Dn
(x entspricht B, W, L)

Dazu gehört das folgende Format des Befehlswortes:

15	14	13	12	11 10 9	8	7 6	5	4	3	2 1 0
1	1	1	0	Zähl-Register	0	Größe	i	0	0	Daten Register

Bit 11..9 Zählregister-Feld
Wenn i = 0:
Die Bits 11.. 9 geben an, um wieviele Positionen die Bits des Zieloperanden nach rechts verschoben werden. Dabei entspricht 001 einer Position usw., bis 111 sieben Positionen entspricht. 000 entspricht aber acht Positionen. (Konstante)
Wenn i = 1:
Die Bits 11..9 wählen ein Datenregister Dn an. Die niederwertigen 6 Bits des Datenregisters Dn geben an, um wieviele Positionen die Bits des Zieloperanden nach rechts verschoben werden.

Bit 7..6 Größe-Feld
00 Byte-Befehl ASR.B
01 Wort-Befehl ASR.W
10 Langwort-Befehl ASR.W

Bit 5 i-Feld
0 Die Bits 11.. 9 beziehen sich auf eine Konstante
1 Die Bits 11.. 9 beziehen sich auf ein Datenregister.

Bit 2..0 Register-Feld
wählt ein Datenregister als Zieloperand an.

Arithmetisches Schieben nach rechts
arithmetic shift right

ASR

Assembler Syntax: ASR <ea>

Die 16 Bits des angewählten Speicherwortes werden um eine Position nach rechts verschoben.

Dazu gehört das folgende Format des Befehlswortes:

15	14	13	12	11	10	9	8	7	6	5 4 3	2 1 0
1	1	1	0	0	0	0	0	1	1	Effektive Adresse Mode	Register

Bit 5..0 Wählt die Effektive Adresse des Quelloperanden an. Die folgenden Adressierungsarten sind erlaubt:

Adr.-Art	Mode	Reg
Dn	nicht erlaubt	
An	nicht erlaubt	
(An)	010	R:An
(An)+	011	R:An
-(An)	100	R:An
d16(An)	101	R:An
d8(An,Xi)	110	R:An

Adr.-Art	Mode	Reg
xxx.W	111	000
xxx.L	111	001
d16(PC)	nicht erlaubt	
d8(PC,Xi)	nicht erlaubt	
#<data>	nicht erlaubt	
Erläuterung siehe Kapitel 5		

Für die Adressierungsart Dn siehe auf der vorherigen Seite.

Benutzen Sie
ASL: wenn nach links geschoben werden soll
LSR: Wenn das hochwertige Bit den Wert 0 erhalten soll.

Siehe auch: LSR, ROL, ROR, ROXL, ROXR, SWAP

Bcc

Springe bedingt
branch conditionally

Bcc ist ein Überbegriff,
siehe Befehlsliste

Wenn (Bedingung = wahr), dann PC + d -> PC

Größe der Adressdifferenz d: Bcc.S 8 Bit
Bcc.W 16 Bit.

Assembler Syntax:
Bcc.S Marke (Marke innerhalb 128 Byte vom PC)
Bcc.W Marke (Marke innerhalb 32 kByte Byte vom PC)

Beschreibung:

Die Bezeichnung Bcc im Kopf dieser Seite ist stellvertretend für die Befehle BCC, BCSf BEQ, BGE, BGT, BHI, BLE, BLS, BLT, BMI, BNE, BPL, BVC, BVS und BRA. Wir fassen alle diese Befehle hier zusammen.

Diese Befehle sind bedingte Sprünge. Das bedeutet, daß der Programmzähler sich zu einem bestimmten Wert ändert, wenn eine bestimmte Bedingung im CCR-Register erfüllt wird.
Wenn die Bedingung nicht erfüllt ist, findet der Sprung nicht statt. Der nächste Befehl ist dann der, der im Speicher direkt auf den Bcc-Befehl folgt.

Der Befehl BRA ist der einzige unbedingte Sprung in der Liste: der Sprung findet immer statt. Wir haben BRA hier mit dazu genommen, weil er das gleiche Format wie die bedingten Sprünge hat.

Springe bedingt
branch conditionally

Bcc

Ist die Adressdifferenz zwischen PC und Marke größer
als 128 Byte für Bcc.S,
oder 32 kByte für Bcc.W,
gibt der Assembler eine Fehlermeldung aus.

Es handelt sich hier bei allen Befehlen um "kurze" Sprünge, relativ zum Programmzähler. Die Ansprung-adresse muß innerhalb eines Bereiches von 64 kByte des Programmzählers liegen.

PROGRAMMIERHINWEIS:

Wenn die erwünschte Ansprungadresse außerhalb dieses Bereiches von 64 kBytes liegt, legen Sie innerhalb des Bereiches eine "Insel" mit einer Zwischen-Ansprungadresse an. Auf der Insel befindet sich dann ein JMP-Befehl zu der von Ihnen erwünschten Ansprungadresse.

Bei den Befehlen haben wir neben den deutschen auch die originalen (amerikanischen) Befehlsbezeichnungen gegeben. Damit können sie sich die Mnemonics besser merken.

Bei den Befehlen werden die Bits N (Negative), Z (Zero), V (oVerflow) und C (Carry) des Condition Code Registers benutzt.

Bcc — **Springe bedingt**
branch conditionally

BCC springe, wenn C-Bit (Carry) zurückgesetzt ist
Branch if Carry is Clear

BCS springe, wenn C-Bit (Carry) gesetzt ist
Branch if Carry is Set

BEQ springe, wenn gleich.
Branch if EQual
Es wird gesprungen, wenn das Z-Bit (Zero) gesetzt ist.

BGE springe, wenn größer oder gleich.
Branch on Greater than or Egual
Es wird gesprungen, wenn das N-Bit (Negative) und das V-Bit (oVerflow) entweder beide gesetzt oder beide zurückgesetzt sind.
BGE ist für Binärzahlen mit Vorzeichen gedacht.

BGT springe, wenn größer
Branche on Greater Than
Es wird gesprungen, wenn
- o das N-Bit und das V-Bit gesetzt sind und das Z-Bit zurückgesetzt ist,

  oder
- o das N-Bit, das V-Bit und das Z-Bit alle zurückgesetzt sind.

BGT ist für Binärzahlen mit Vorzeichen gedacht, ist sonst ähnlich BHI.

BHI springe, wenn höher
Branch on Higher than
Es wird gesprungen, wenn das C-Bit und das Z-Bit beide zurückgesetzt sind.
BGE ist für Binärzahlen ohne Vorzeichen gedacht, ist sonst ähnlich BGT.

Springe bedingt
branch conditionally

Bcc

BLE springe, wenn kleiner oder gleich
Branch on Less than or Egual

Es wird gesprungen, wenn
- o das Z-Bit gesetzt ist,
  oder
- o das N-Bit gesetzt und das V-Bit zurückgesetzt ist,
  oder
- o das N-Bit zurückgesetzt und das V-Bit gesetzt ist.

BLE ist für Binärzahlen mit Vorzeichen gedacht, ist sonst ähnlich BLS.

BLS springe, wenn niedriger oder gleich
Branch on Lower or Same

Es wird gesprungen, wenn das C-Bit, das Z-Bit oder beide gesetzt sind.
BLS ist für Binärzahlen ohne Vorzeichen gedacht, ist sonst ähnlich BLE.

BLT springe, wenn kleiner
Branch on Less Than

Es wird gesprungen, wenn
- o das N-Bit gesetzt und das V-Bit zurückgesetzt,
  oder
- o das N-Bit zurückgesetzt und das V-Bit gesetzt ist.

BLT ist für Binärzahlen mit Vorzeichen gedacht.

BMI springe, wenn Minus
Branch on Minus

Es wird gesprungen, wenn das N-Bit gesetzt ist.
BMI ist für Binärzahlen mit Vorzeichen gedacht.

Bcc

Springe bedingt
branch conditionally

BNE springe, wenn ungleich
Branch on Not Equal
Es wird gesprungen, wenn das Z-Bit zurückgesetzt ist.

BPL springe, wenn Plus
Branch on PLus
Es wird gesprungen, wenn das N-Bit zurückgesetzt ist.
BPL ist für Binärzahlen mit Vorzeichen gedacht.

BVC springe, wenn kein Überlauf
Branch on oVerflow Clear
Es wird gesprungen, wenn das V-Bit zurückgesetzt ist.

BVS springe, wenn Überlauf
Branch on oVerflow Set
Es wird gesprungen, wenn das V-Bit gesetzt ist.

BRA springe unbedingt
BRanch Always
Es wird immer gesprungen, unabhängig vom CCR.

Springe bedingt
branch conditionally

Bcc

Übersicht der Sprung-Bedingungen

N	Z	V	C	BCC	BCS	BEQ	BGE	BGT	BHI	BLE	BLS	BLT	BMI	BNE	BPL	BVC	BVS	BRA
0	0	0	0	+	-	-	+	+	+	-	-	-	-	+	+	+	-	+
0	0	0	1	-	+	-	+	+	-	-	+	-	-	+	+	+	-	+
0	0	1	0	+	-	-	-	-	+	+	-	+	-	+	+	-	+	+
0	0	1	1	-	+	-	-	-	-	+	+	+	-	+	+	-	+	+
0	1	0	0	+	-	+	+	-	-	+	+	-	-	-	+	+	-	+
0	1	0	1	-	+	+	+	-	-	+	+	-	-	-	+	+	-	+
0	1	1	0	+	-	+	-	-	-	+	+	+	-	-	+	-	+	+
0	1	1	1	-	+	+	-	-	-	+	+	+	-	-	+	-	+	+
1	0	0	0	+	-	-	-	-	+	+	-	+	+	+	-	+	-	+
1	0	0	1	-	+	-	-	-	-	+	+	+	+	+	-	+	-	+
1	0	1	0	+	-	-	+	+	+	-	-	-	+	+	-	-	+	+
1	0	1	1	-	+	-	+	+	-	-	+	-	+	+	-	-	+	+
1	1	0	0	+	-	+	-	-	-	+	+	+	+	-	-	+	-	+
1	1	0	1	-	+	+	-	-	-	+	+	+	+	-	-	+	-	+
1	1	1	0	+	-	+	+	-	-	+	+	-	+	-	-	-	+	+
1	1	1	1	-	+	+	+	-	-	+	+	-	+	-	-	-	+	+

+ bedeutet: der Sprung findet statt
- bedeutet: der Sprung findet nicht statt

X	N	Z	V	C
-	-	-	-	-

Condition Code Register:
bleibt unverändert



Springe bedingt
branch conditionally

Format des Befehlswortes:

15	14	13	12	11 10 9 8	7 6 5 4 3 2 1 0
1	1	0	1	Bedingung	8-Bit Adressdifferenz
16-Bit Adressdiferenz, wenn der 8-Bit Adressdifferenz Null ist					

Bit 11..8 Bedingungsfeld

Bedingung	Befehl	Bedingung	Befehl
0000	BRA	1000	BVC
0001	(kein)	1001	BVS
0010	BHI	1010	BPL
0011	BLS	1011	BMI
0100	BCC	1100	BGE
0101	BCS	1101	BLT
0110	BNE	1110	BGT
0111	BEQ	1111	BLE

Bemerkungen:
1. Es gibt keine Bcc- Bedingung, so daß der Sprung NIE stattfindet. Nehmen Sie dazu NOP ($4E71)
2. Die Zahl 0001 im Bedingungsfeld entspricht keinem bedingten Sprung, sondern einem BSR-Befehl (springe zum Unterprogramm).

Feld Adress-differenz:
wird als Binärzahl mit Vorzeichen aufgefaßt. Wenn die Bedingung erfüllt ist, findet der Sprung zu der Adresse (PC + 8-Bit Adress-Differenz) statt. Wenn alle Bits des Feldes 8-Bit Adress-differenz Null sind, wird als Operand das Feld 16-Bit Adress-differenz genommen.

Prüfe Bit und ändere
test a bit and change **BCHG**

(angewähltes Bit des Operanden) -> Z-Bit
(angewähltes Bit des Operanden) ->
angewähltes Bit des Operanden

Operandgröße: BCHG.B Byte (8 Bit)
BCHG.L Langwort (32 Bit)

Assembler Syntax: BCHG.L #<data>, Dn
BCHG.L Dp, Dn
BCHG.B #<data>, <ea>
BCHG.B Dp, <ea>
(Bit-Zeiger, Operand)

Beschreibung:

Mit BCHG können Sie die einzelnen Bits des Operanden direkt abfragen. Ergebnis:

getestetes Bit = 0 -> Bit Z wird gesetzt
getestetes Bit = 1 -> Bit Z wird zurückgesetzt

Das getestete Bit wird anschließend geändert:
wenn es gesetzt ist, wird es gelöscht,
wenn es gelöscht ist, wird es gesetzt.

X	N	Z	V	C
-	-	*	-	-

Condition Code Register:
bleibt unverändert

C bleibt unverändert.
V bleibt unverändert.
Z wird gesetzt, wenn das getestete Bit Null ist. Wird sonst zurückgesetzt.
N bleibt unverändert.
X bleibt unverändert.

BCHG

Prüfe Bit und ändere
test a bit and change

Es gibt vier Befehlsformen.

- o Der Befehl **BCHG.L #<data>, Dn**
  prüft ein Bit des Datenregisters Dn.
  Die Konstante #<data> wählt das Bit an.

- o Der Befehl **BCHG.L Dp, Dn**
  prüft ein Bit des Datenregisters Dn.
  Das Datenregister Dp wählt das Bit an.

- o Der Befehl **BCHG.B #<data>, <ea>**
  prüft ein Bit in einem Byte im Speicher.
  Die Konstante #<data> wählt das Bit an.

- o Der Befehl **BCHG.B Dp, <ea>**
  prüft ein Bit in einem Byte im Speicher.
  Das Datenregister Dp wählt das Bit an.

Assembler Syntax: BCHG.L #<data>, Dn

Dazu gehört das folgende Format des Befehlswortes:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	0	0	0	0	1	0	0	0	Register Dn		
								B i t z a h l							

Bit 2..0 Feld Datenregister Dn
Wählt das Datenregister mit dem Operand an. Hierin befindet sich das Bit, das geprüft werden soll.

Feld Bitzahl
gibt an, welches Bit von Dn geprüft wird.
Es werden nur die Bits 0..4 der Bitzahl benutzt.
Es wird also Bit[(Bitzahl) mod 32] von Dn geprüft.

Prüfe Bit und ändere
test a bit and change

BCHG

Assembler Syntax: BCHG.L Dp, Dn

Dazu gehört das folgende Format des Befehlswortes:

```
| 15  14  13  12| 11  10  9   8 | 7   6   5   4 | 3   2   1   0 |
|   |   |   |   |             |   |   |   |   |   |   |           |
| 0 | 0 | 0 | 0 | Register    | 1 | 0 | 1 | 0 | 0 | 0 | Register  |
|   |   |   |   |    Dp       |   |   |   |   |   |   |    Dn     |
```

Bit 11..9 Feld Datenregister Dp
gibt an, welches Bit von Dn geprüft wird.
Es werden nur die Bits 0..4 von Dp benutzt.
Es wird also Bit[(Dp) mod 32] von Dn geprüft.

Bit 2..0 Feld Datenregister Dn
Wählt das Datenregister mit dem Operand an. Hierin
befindet sich das Bit, das geprüft werden soll.

BCHG

Prüfe Bit und ändere
test a bit and change

Assembler Syntax: BCHG.B #<data>, <ea>

Dazu gehört das folgende Format des Befehlswortes:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	1	Effektive Adresse Mode			Register		
								Bitzahl							

Bit 5..0 wählt die Effektive Adresse des Operanden
Die folgenden Adressierungsarten sind erlaubt:

Adr.-Art	Mode	Reg
Dn	nicht erlaubt	
An	nicht erlaubt	
(An)	010	R:An
(An)+	011	R:An
-(An)	100	R:An
d16(An)	101	R:An
d8(An,Xi)	110	R:An

Adr.-Art	Mode	Reg
xxx.W	111	000
xxx.L	111	001
d16(PC)	111	010
d8(PC,Xi)	111	011
#<data>	nicht erlaubt	
Erläuterung siehe Kapitel 5		

Feld Bitzahl
gibt an, welches Bit des Operanden geprüft wird.
Es werden nur die Bits 0..2 der Bitzahl benutzt.
Es wird also Bit[(Bit Zahl) mod 8] des Operanden geprüft.

Prüfe Bit und ändere
test a bit and change

BCHG

Assembler Syntax: BCHG.B Dp, <ea>

Dazu gehört das folgende Format des Befehlswortes:

15	14	13	12	11 10 9	8	7	6	5 4 3	2 1 0
0	0	0	0	Register Dp	1	0	1	Effektive Adresse Mode	Effektive Adresse Register

Bit 11..9 Feld Datenregister Dp
Gibt an, welches Bit des Operanden geprüft wird.
Es werden nur die Bits 0..2 dieses Registers benutzt,
Es wird also Bit[(Bit Zahl) mod 8] des Operanden
geprüft.

Bit 5..0 wählt die Effektive Adresse des Operanden
an. Die folgenden Adressierungsarten sind erlaubt:

Adr.-Art	Mode	Reg
Dn	nicht erlaubt	
An	nicht erlaubt	
(An)	010	R:An
(An)+	011	R:An
-(An)	100	R:An
d16(An)	101	R:An
d8(An,Xi)	110	R:An

Adr.-Art	Mode	Reg
xxx.W	111	000
xxx.L	111	001
d16(PC)	111	010
d8(PC,Xi)	111	011
#<data>	111	100
Erläuterung siehe Kapitel 5		

Benutzen Sie wenn das angewählte Bit
BTST: nur getestet werden soll
BCLR: getestet und gelöscht werden soll
BSET: getestet und gesetzt werden soll

BCLR

Prüfe Bit und lösche
test a bit and clear

~(angewähltes Bit des Operanden) -> Z-Bit
0 -> angewähltes Bit des Operanden

Operandgröße: BCLR.B Byte (8 Bit)
BCLR.L Langwort (32 Bit)

Assembler Syntax:
BCLR.L #<data>, Dn
BCLR.L Dp, Dn
BCLR.B #<data>, <ea>
BCLR.B Dp, <ea>
(Bit-Zeiger, Operand)

Beschreibung:
Mit BCLR können Sie die einzelnen Bits des Operanden direkt abfragen. Ergebnis:

getestetes Bit = 0 -> Bit Z wird gesetzt
getestetes Bit = 1 -> Bit Z wird zurückgesetzt

Das getestete Bit wird anschließend gelöscht.

X	N	Z	V	C
-	-	*	-	-

Condition Code Register:
bleibt unverändert

C bleibt unverändert.
V bleibt unverändert.
Z wird gesetzt, wenn das getestete Bit Null ist.
Wird sonst zurückgesetzt.
N bleibt unverändert.
X bleibt unverändert.

Prüfe Bit und lösche
test a bit and clear

BCLR

Es gibt vier Befehlsformen.

- o Der Befehl **BCLR.L #<data>, Dn**
  prüft ein Bit des Datenregisters Dn.
  Die Konstante #<data> wählt das Bit an.

- o Der Befehl **BCLR.L Dp, Dn**
  prüft ein Bit des Datenregisters Dn.
  Das Datenregister Dp wählt das Bit an.

- o Der Befehl **BCLR.B #<data>, <ea>**
  prüft ein Bit in einem Byte im Speicher.
  Die Konstante #<data> wählt das Bit an.

- o Der Befehl **BCLR.B Dp, <ea>**
  prüft ein Bit in einem Byte im Speicher.
  Das Datenregister Dp wählt das Bit an.

Assembler Syntax: BCLR.L #<data>, Dn

Dazu gehört das folgende Format des Befehlswortes:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	1	0	0	0	0	Register Dn		
								B i t z a h l							

Bit 2..0 Feld Datenregister Dn
Wählt das Datenregister mit dem Operand an. Hierin befindet sich das Bit, das geprüft werden soll.

Feld Bitzahl
gibt an, welches Bit von Dn geprüft wird.
Es werden nur die Bits 0..4 der Bitzahl benutzt.
Es wird also Bit[(Bitzahl) mod 32] von Dn geprüft.

BCLR

Prüfe Bit und lösche
test a bit and clear

Assembler Syntax: BCLR.L Dp, Dn

Dazu gehört das folgende Format des Befehlswortes:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	Register Dp			1	1	0	0	0	0	Register Dn		

Bit 11..9 Feld Datenregister Dp
gibt an, welches Bit von Dn geprüft wird.
Es werden nur die Bits 0..4 von Dp benutzt.
Es wird also Bit[(Dp) mod 32] von Dn geprüft.

Bit 2..0 Feld Datenregister Dn
Wählt das Datenregister mit dem Operand an. Hierin befindet sich das Bit, das geprüft werden soll.

Prüfe Bit und lösche
test a bit and clear

BCLR

Assembler Syntax: BCLR.B #<data>, <ea>

Dazu gehört das folgende Format des Befehlswortes:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	1	0	Effektive Adresse Mode			Effektive Adresse Register		
								B i t z a h l							

Bit 5..0 wählt die Effektive Adresse des Operanden an. Die folgenden Adressierungsarten sind erlaubt:

Adr.-Art	Mode	Reg
Dn	nicht erlaubt	
An	nicht erlaubt	
(An)	010	R:An
(An)+	011	R:An
-(An)	100	R:An
d16(An)	101	R:An
d8(An,Xi)	110	R:An

Adr.-Art	Mode	Reg
xxx.W	111	000
xxx.L	111	001
d16(PC)	111	010
d8(PC,Xi)	111	011
#<data>	nicht erlaubt	
Erläuterung siehe Kapitel 5		

Feld Bitzahl
gibt an, welches Bit des Operanden geprüft wird. Es werden nur die Bits 0..2 der Bitzahl benutzt. Es wird also Bit[(Bit Zahl) mod 8] des Operanden geprüft.

BCLR

Prüfe Bit und lösche
test a bit and clear

Assembler Syntax: BCLR.B Dp, <ea>

Dazu gehört das folgende Format des Befehlswortes:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	Register Dp			1	1	0	Effektive Adresse Mode			Register		

Bit 11..9 Feld Datenregister Dp
Gibt an, welches Bit des Operanden geprüft wird.
Es werden nur die Bits 0..2 dieses Registers benutzt,
Es wird also Bit[(Bit Zahl) mod 8] des Operanden
geprüft.

Bit 5..0 wählt die Effektive Adresse des Operanden an. Die folgenden Adressierungsarten sind erlaubt:

Adr.-Art	Mode	Reg
Dn	nicht erlaubt	
An	nicht erlaubt	
(An)	010	R:An
(An)+	011	R:An
-(An)	100	R:An
d16(An)	101	R:An
d8(An,Xi)	110	R:An

Adr.-Art	Mode	Reg
xxx.W	111	000
xxx.L	111	001
d16(PC)	111	010
d8(PC,Xi)	111	011
#<data>	111	100
Erläuterung siehe Kapitel 5		

Benutzen Sie wenn das angewählte Bit
BTST: nur getestet werden soll
BSET: getestet und gesetzt werden soll
BCHG: getestet und geändert werden soll

Prüfe Bit und setze
test a bit and set

BSET

~(angewähltes Bit des Operanden) -> Z-Bit
1 -> angewähltes Bit des Operanden

Operandgröße: BSET.B Byte (8 Bit)
BSET.L Langwort (32 Bit)

Assembler Syntax:

```
BSET.L  #<data>, Dn
BSET.L  Dp, Dn
BSET.B. #<data>, <ea>
BSET.B  Dp, <ea>
           (Bit-Zeiger, Operand)
```

Beschreibung:

Mit BSET können Sie die einzelnen Bits des Operanden direkt abfragen. Ergebnis:

getestetes Bit = 0 -> Bit Z wird gesetzt
getestetes Bit = 1 -> Bit Z wird zurückgesetzt

Das getestete Bit wird anschließend gesetzt.

Condition Code Register:

X	N	Z	V	C
-	-	*	-	-

C bleibt unverändert.
V bleibt unverändert.
Z wird gesetzt, wenn das getestete Bit Null ist. Wird sonst zurückgesetzt.
N bleibt unverändert.
X bleibt unverändert.

BSET

Prüfe Bit und setze
test a bit and set

Es gibt vier Befehlsformen.

- o Der Befehl **BSET.L #<data>, Dn**
  prüft ein Bit des Datenregisters Dn.
  Die Konstante #<data> wählt das Bit an.

- o Der Befehl **BSET.L Dp, Dn**
  prüft ein Bit des Datenregisters Dn.
  Das Datenregister Dp wählt das Bit an.

- o Der Befehl **BSET.B #<data>, <ea>**
  prüft ein Bit in einem Byte im Speicher.
  Die Konstante #<data> wählt das Bit an.

- o Der Befehl **BSET.B Dp, <ea>**
  prüft ein Bit in einem Byte im Speicher.
  Das Datenregister Dp wählt das Bit an.

Assembler Syntax: BSET.L #<data>, Dn

Dazu gehört das folgende Format des Befehlswortes:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	1	1	0	0	0	Register Dn		
								B i t z a h l							

Bit 2..0 Feld Datenregister Dn
Wählt das Datenregister mit dem Operand an. Hierin befindet sich das Bit, das geprüft werden soll.

Feld Bitzahl
gibt an, welches Bit von Dn geprüft wird.
Es werden nur die Bits 0..4 der Bitzahl benutzt.
Es wird also Bit[(Bitzahl) mod 32] von Dn geprüft.

Prüfe Bit und setze
test a bit and set

BSET

Assembler Syntax: BSET.L Dp, Dn

Dazu gehört das folgende Format des Befehlswortes:

15	14	13	12	11 10 9	8	7	6	5	4	3	2 1 0
0	0	0	0	Register Dp	1	1	1	0	0	0	Register Dn

Bit 11..9 Feld Datenregister Dp
gibt an, welches Bit von Dn geprüft wird.
Es werden nur die Bits 0..4 von Dp benutzt.
Es wird also Bit[(Dp) mod 32] von Dn geprüft.

Bit 2..0 Feld Datenregister Dn
Wählt das Datenregister mit dem Operand an. Hierin
befindet sich das Bit, das geprüft werden soll.

BSET

Prüfe Bit und setze
test a bit and set

Assembler Syntax: BSET.B #<data>f <ea>

Dazu gehört das folgende Format des Befehlswortes:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	1	1	Effektive Adresse Mode			Register		
								B i t z a h l							

Bit 5..0 wählt die Effektive Adresse des Operanden an. Die folgenden Adressierungsarten sind erlaubt:

Adr.-Art	Mode	Reg
Dn	nicht erlaubt	
An	nicht erlaubt	
(An)	010	R:An
(An)+	011	R:An
-(An)	100	R:An
d16(An)	101	R:An
d9(An,Xi)	110	R:An

Adr.-Art	Mode	Reg
xxx.W	111	000
xxx.L	111	001
d16(PC)	111	010
d8(PC,Xi)	111	011
#<data>	nicht erlaubt	
Erläuterung siehe Kapitel 5		

Feld Bitzahl
gibt an, welches Bit des Operanden geprüft wird. Es werden nur die Bits 0..2 der Bitzahl benutzt. Es wird also Bit[(Bit Zahl) mod 8] des Operanden geprüft.

Prüfe Bit und setze
test a bit and set

BSET

Assembler Syntax: BSET.B Dp, <ea>

Dazu gehört das folgende Format des Befehlswortes:

15	14	13	12	11 10 9	8	7	6	5 4	3 2 1 0
0	0	0	0	Register Dp	1	1	1	Effektive Adresse Mode	Effektive Adresse Register

Bit 11..9 Feld Datenregister Dp
Gibt an, welches Bit des Operanden geprüft wird.
Es werden nur die Bits 0..2 dieses Registers benutzt
Es wird also Bit[(Bit Zahl) mod 8] des Operanden
geprüft.

Bit 5..0 wählt die Effektive Adresse des Operanden
an. Die folgenden Adressierungsarten sind erlaubt:

Adr.-Art	Mode	Reg
Dn	nicht erlaubt	
An	nicht erlaubt	
(An)	010	R:An
(An)+	011	R:An
-(An)	100	R:An
d16(An)	101	R:An
d8(An,Xi)	110	R:An

Adr.-Art	Mode	Reg
xxx.W	111	000
xxx.L	111	001
d16(PC)	111	010
d8(PC,Xi)	111	011
#<data>	111	100
Erläuterung siehe Kapitel 5		

Benutzen Sie	wenn das angewählte Bit
BTST:	nur getestet werden soll
BCLR:	getestet und gelöscht werden soll
BCHG:	getestet und geändert werden soll



Aufruf Unterprogramm (relativ)
branch to subroutine

SP - 4 -> SP; PC -> (SP); PC + d -> PC

Größe der Adressdifferenz d: BSR.S 8 Bit
BSR.W 16 Bit.

Assembler Syntax:

```
BSR.S  Marke   (Marke innerhalb 128 Byte vom PC)
BSR.W  Marke   (Marke innerhalb 32 kByte Byte vom PC)
```

Beschreibung:

Die 32-Bit Adresse des Befehls, der im Speicher direkt auf den BSR-Befehl folgt, wird auf den Stack gepushed.

Danach geht die Ausführung des Programms weiter bei der Marke.

Ist die Adressdifferenz zwischen PC und Marke größer
als 128 Byte für BSR.S,
oder 32 kByte für BSR.W,
gibt der Assembler eine Fehlermeldung aus.

PROGRAMMIERHINWEIS:

Beenden Sie das Unterprogramm mit RTS.
Die Programmausfuhr geht danach weiter mit dem Befehl, der im Speicher direkt auf den BSR-Befehl folgt.

Aufruf Unterprogramm (relativ)
branch to subroutine

BSR

X	N	Z	V	C
-	-	-	-	-

Condition Code Register:
bleibt unverändert

Format des Befehlswortes:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	0	0	1	8-Bit Adressdifferenz							
16-Bit Adressdiferenz, wenn der 8-Bit Adressdifferenz Null ist															

Feld Adress-differenz:
wird als Binärzahl mit Vorzeichen aufgefaßt.
Wenn die Bedingung erfüllt ist, findet der Sprung zur Adresse (PC + 8-Bit Adress-Differenz) statt.
Wenn alle Bits des Feldes 8-Bit Adress-differenz Null sind, wird als Operand das Feld 16-Bit Adressdifferenz genommen.

Benutzen Sie

JSR:	wenn die Sprungadresse außerhalb des Bereichs von 32 kByte liegt.
JMP oder BRA:	wenn keinen Rückkehr erwünscht ist.

BTST

Prüfe Bit
test a bit

~(angewähltes Bit des Operanden) -> Z-Bit

Operandgröße: BTST.B Byte (8 Bit)
BTST.L Langwort (32 Bit)

Assembler Syntax:

```
BTST.L  #<data>, Dn
BTST.L  Dp, Dn
BTST.B  #<data>, <ea>
BTST.B  Dp, <ea>
           (Bit-Zeiger, Operand)
```

Beschreibung:

Mit BTST können Sie die einzelnen Bits des Operanden direkt abfragen. Ergebnis:

getestetes Bit = 0 -> Bit Z wird gesetzt
getestetes Bit = 1 -> Bit Z wird zurückgesetzt

Das getestete Bit bleibt unverändert.

Condition Code Register:

X	N	Z	V	C
-	-	*	-	-

C bleibt unverändert.
V bleibt unverändert.
Z wird gesetzt, wenn das getestete Bit Null ist. Wird sonst zurückgesetzt.
N bleibt unverändert.
X bleibt unverändert.

Prüfe Bit
test a bit

BTST

Es gibt vier Befehlsformen.

- o Der Befehl **BTST.L #<data>, Dn**
  prüft ein Bit des Datenregisters Dn.
  Die Konstante #<data> wählt das Bit an.

- o Der Befehl **BTST.L Dp, Dn**
  prüft ein Bit des Datenregisters Dn.
  Das Datenregister Dp wählt das Bit an.

- o Der Befehl **BTST.B #<data>, <ea>**
  prüft ein Bit in einem Byte im Speicher.
  Die Konstante #<data> wählt das Bit an.

- o Der Befehl **BTST.B Dp, <ea>**
  prüft ein Bit in einem Byte im Speicher.
  Das Datenregister Dp wählt das Bit an.

Assembler Syntax: BTST.L #<data>, Dn

Dazu gehört das folgende Format des Befehlswortes:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	0	0	0	0	Register Dn		
								B i t z a h l							

Bit 2..0 Feld Datenregister Dn
Wählt das Datenregister mit dem Operand an. Hierin befindet sich das Bit, das geprüft werden soll.

Feld Bitzahl
gibt an, welches Bit von Dn geprüft wird.
Es werden nur die Bits 0..4 der Bitzahl benutzt.
Es wird also Bit[Bitzahl mod 32] von Dn geprüft.

BTST

Prüfe Bit
test a bit

Assembler Syntax: BTST.L Dp, Dn

Dazu gehört das folgende Format des Befehlswortes:

15	14	13	12	11 10 9	8	7	6	5	4	3	2 1 0
0	0	0	0	Register Dp	1	0	0	0	0	0	Register Dn

Bit 11..9 Feld Datenregister Dp
gibt an, welches Bit von Dn geprüft wird.
Es werden nur die Bits 0..4 von Dp benutzt.
Es wird also Bit[(Dp) mod 32] von Dn geprüft.

Bit 2..0 Feld Datenregister Dn
Wählt das Datenregister mit dem Operand an. Hierin befindet sich das Bit, das geprüft werden soll.

Prüfe Bit
test a bit

BTST

Assembler Syntax: BTST.B #<data>, <ea>

Dazu gehört das folgende Format des Befehlswortes:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	0	Effektive Adresse Mode			Register		
								B i t z a h l							

Bit 5..0 wählt die Effektive Adresse des Operanden an. Die folgenden Adressierungsarten sind erlaubt:

Adr.-Art	Mode	Reg
Dn	nicht erlaubt	
An	nicht erlaubt	
(An)	010	R:An
(An)+	011	R:An
-(An)	100	R:An
d16(An)	101	R:An
d8(An,Xi)	110	R:An

Adr.-Art	Mode	Reg
xxx.W	111	000
xxx.L	111	001
d16(PC)	111	010
d8(PC,Xi)	111	011
#<data>	nicht erlaubt	
Erläuterung siehe Kapitel 5		

Feld Bitzahl
gibt an, welches Bit des Operanden geprüft wird. Es werden nur die Bits 0..2 der Bitzahl benutzt. Es wird also Bit[(Bit Zahl) mod 8] des Operanden geprüft.

BTST

Prüfe Bit
test a bit

Assembler Syntax: BTST.B Dp, <ea>

Dazu gehört das folgende Format des Befehlswortes:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	Register Dp			1	0	0	Effektive Adresse Mode			Register		

Bit 11..9 Feld Datenregister Dp
Gibt an, welches Bit des Operanden geprüft wird.
Es werden nur die Bits 0..2 dieses Registers benutzt.
Es wird also Bit[(Bit Zahl) mod 8] des Operanden geprüft.

Bit 5..0 wählt die Effektive Adresse des Operanden an. Die folgenden Adressierungsarten sind erlaubt:

Adr.-Art	Mode	Reg
Dn	nicht erlaubt	
An	nicht erlaubt	
(An)	010	R:An
(An)+	011	R:An
-(An)	100	R:An
d16(An)	101	R:An
d8(An,Xi)	110	R:An

Adr.-Art	Mode	Reg
xxx.W	111	000
xxx.L	111	001
d16(PC)	111	010
d8(PC,Xi)	111	011
#<data>	111	100
Erläuterung siehe Kapitel 5		

Benutzen Sie	wenn das angewählte Bit
BCHG:	getestet und geändert werden soll
BCLR:	getestet und gelöscht werden soll
BSET:	getestet und gesetzt werden soll

Vergleiche Register mit Grenzen
check register against bounds **CHK**

Wenn Dn < Null oder Dn > Quelle, dann wird Exception 6 ausgelöst.

Operandgröße: Wort (16 Bit)

Assembler Syntax: CHK <ea>, Dn
(Quelle, Ziel)

Beschreibung:

Das niederwertige Wort des angewählten Datenregisters (Bit 15..0) wird geprüft und mit dem Quelloperand verglichen. Wenn der Registerwert negativ ist (also Bit 15 gesetzt), oder den Grenzwert überschreitet, dann wird die Exception 6 ausgelöst, siehe Kap. 6.

Wenn der Registerwert aber zwischen Null und dem Grenzwert liegt, findet keine Aktion statt. Der nächste Befehl im Speicher wird dann ausgeführt.

Beide Argumente werden als Binärzahl mit Vorzeichen aufgefaßt.

Condition Code Register:

X	N	Z	V	C
-	*	U	U	U

C nicht definiert.
V nicht definiert.
Z nicht definiert.
N wird gesetzt wenn Dn < Null.
wird zurückgesetzt wenn Dn > Quelle.
Ist sonst nicht definiert.
X Ändert sich nicht.

CHK — **Vergleiche Register mit Grenzen**
check register against bounds

Vergleiche Register mit Grenzen
check register against bounds — **CHK**

X	N	Z	V	C
*	*	*	*	*

Condition Code Register:

Format des Befehlswortes:

15	14	13	12	11 10 9	8	7	6	5 4 3 2 1 0
0	1	0	0	Register Dp	1	1	0	Effektive Adresse Mode | Register

Bit 11..9 Registerfeld: wählt eines der prüfenden Datenregister an.

Bit 5..0 wählt die obere Grenze an. Die folgenden Adressierungsarten sind erlaubt:

Adr.-Art	Mode	Reg
Dn	000	R:Dn
An	nicht erlaubt	
(An)	010	R:An
(An)+	011	R:An
-(An)	100	R:An
d16(An)	101	R:An
d8(An,Xi)	110	R:An

Adr.-Art	Mode	Reg
xxx.W	111	000
xxx.L	111	001
d16(PC)	111	010
d8(PC,Xi)	111	011
#<data>	111	100
Erläuterung siehe Kapitel 5		

Benutzen Sie TRAPV: wenn ein gesetztes V-Bit eine Exception auslösen soll.

Lösche Operand
clear an operand

CLR

0 -> Ziel

```
Operandgröße:   CLR.B  Byte (8 Bit)
                CLR.W  Wort (16 Bit)
                CLR.L  Langwort (32 Bit.)

Assembler       CLR.x  <ea>
Syntax:            (x entspricht B, W, L)
```

Beschreibung:

Das Ziel erhält den Wert Null.

Der Operandgröße kann ein Byte, ein Wort oder ein Langwort sein.

Condition Code Register:

X	N	Z	V	C
-	0	1	0	0

C wird zurückgesetzt
V wird zurückgesetzt
Z wird gesetzt.
N wird zurückgesetzt,
X bleibt unverändert

CLR

Lösche Operand
clear an operand

Format des Befehlswortes:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	1	0	Größe		Effektive Adresse Mode			Register		

Bit 7..6 Größe-Feld:
00 Byte-Befehl CLR.B
01 Wort-Befehl CLR.W
10 Langwort-Befehl CLR.L

Aufbau der Argumentwörter siehe Kap. 3.8

Bit 5..0 Die Effektive Adresse wählt den Zieloperand an. Die folgenden Adressierungsarten sind erlaubt:

Adr.-Art	Mode	Reg
Dn	000	R:Dn
An	nicht erlaubt	
(An)	010	R:An
(An)+	011	R:An
-(An)	100	R:An
d16(An)	101	R:An
d8(An,Xi)	110	R:An

Adr.-Art	Mode	Reg
xxx.W	111	000
xxx.L	111	001
d16(PC)	nicht erlaubt	
d8(PC,Xi)	nicht erlaubt	
#<data>	nicht erlaubt	
Erläuterung siehe Kapitel 5		

Argumentwort:
Bit 7..6 = 00 -> Datenfeld ist die niederwertige Hälfte des 1. Argumentwortes
Bit 7..6 = 01 -> Datenfeld ist das 1. Argumentwort
Bit 7..6 = 10 -> Datenfeld ist 1. + 2. Argumentwort

Siehe auch: MOVEQ, MOVE

Vergleiche
compare

CMP

Ziel - Quelle

Operandgröße: CMP.B Byte (8 Bit)
CMP.W Wort (16 Bit)
CMP.L Langwort (32 Bit)

Assembler Syntax: CMP.x <ea>, Dn (Quelle,Ziel)
(x entspricht B, W, L)

Beschreibung:

Der Quelloperand wird vom angegebenen Datenregister subtrahiert, und das Condition Code Register wird entsprechend gesetzt. Das Datenregister wird nicht geändert.

Der Operandgröße kann ein Byte, ein Wort oder ein Langwort sein.

Condition Code Register:

X	N	Z	V	C
-	*	*	*	*

C wird gesetzt, wenn ein "Leihen" generiert wird. Wird sonst zurückgesetzt.
V wird gesetzt, wenn ein Überlauf generiert wird. Wird sonst zurückgesetzt.
Z wird gesetzt, wenn das Ergebnis gleich Null ist. Wird sonst zurückgesetzt.
N wird gesetzt, wenn das Ergebnis negativ ist. Wird sonst zurückgesetzt.
X bleibt unverändert.

CMP

Vergleiche
compare

Format des Befehlswortes:

```
| 15  14  13  12| 11  10  9   8 | 7   6   5   4 | 3   2   1   0 |
|   |   |   |   |             |             |                   |
| 1 | 0 | 1 | 1 |  Register   |Operations-|     Effektive Adresse |
|   |   |   |   |     Dp      |    Mode    |    Mode    |   Register|
```

Bit 11..9 Registerfeld: wählt eines der acht Datenregister als Ziel an.

Bit 8..6 Feld Operationsmode:

CMP.B	CMP.W	CMP.L	Operation
000	001	010	Dn - <ea>

Bit 5..0 Wenn die Effektive Adresse der Quelloperand ist, sind die folgenden Adressierungsarten erlaubt:

Adr.-Art	Mode	Reg
Dn	000	R:Dn
An*)	001	R:An
(An)	010	R:An
(An)+	011	R:An
-(An)	100	R:An
d16(An)	101	R:An
d8(An,Xi)	110	R:An

Adr.-Art	Mode	Reg
xxx.W	111	000
xxx.L	111	001
d16(PC)	111	010
d8(PC,Xi)	111	011
#<data>	111	100
Erläuterung siehe Kapitel 5		

*) Adressierungsart An nicht für Byte-Befehle erlaubt

Benutzen Sie

CMPA: wenn der Zieloperand ein Adressregister ist;
CMPI: wenn einer der Operanden eine Konstante ist.
CMPM: wenn ein direkter Speicher zu Speicher Vergleich stattfinden soll.

Vergleiche Adresse
compare address

CMPA

Ziel - Quelle

Operandgröße: CMPA.W Wort (16 Bit)
CMPA.L Langwort (32 Bit)

Assembler Syntax: CMPA.x <ea>, An (Quelle, Ziel)
(x entspricht W, L)

Beschreibung:
Der Quelloperand in <ea> wird binär vom Ziel-Adressregister An subtrahiert, und das Condition Code Register wird entsprechend gesetzt. Das Ziel-Adressregister An wird nicht geändert.

Wenn die Operandgröße des Quelloperanden ein Wort ist, wird der Quellopperand mit dem gleichen Vorzeichen auf 32 Bit erweitert. Vom Ziel-Adressregister werden sämtliche 32 Bits angewendet.

Condition Code Register:

X	N	Z	V	C
-	*	*	*	*

C wird gesetzt, wenn ein "Leihen" generiert wird. Wird sonst zurückgesetzt.
V wird gesetzt, wenn ein Überlauf generiert wird. Wird sonst zurückgesetzt.
Z wird gesetzt, wenn das Ergebnis gleich Null ist Wird sonst zurückgesetzt.
N wird gesetzt, wenn das Ergebnis negativ ist. Wird sonst zurückgesetzt.
X bleibt unverändert.

CMPA

Vergleiche Adresse
compare address

Format des Befehlswortes:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	Register Dp			Operations-Mode			Effektive Adresse Mode			Register		

Bit 11..9 Registerfeld: wählt eines der acht Adressregister An an. Es ist der Zieloperand.

Bit 8..6 Feld Operations-Mode:

011 CMPA.W - Wort-Befehl. Der Quellopperand wird mit dem gleichen Vorzeichen auf 32 Bit erweitert, und vom Ziel-Adressregister werden sämtliche 32 Bits angewendet.

111 CMPA.L - Langwort-Befehl

Bit 5..0 Die Effektive Adresse wählt den Quelloperand an. Alle Adressierungsarten sind erlaubt:

Adr.-Art	Mode	Reg
Dn	000	R:Dn
An	001	R:An
(An)	010	R:An
(An)+	011	R:An
-(An)	100	R:An
d16(An)	101	R:An
d8(An,Xi)	110	R:An

Adr.-Art	Mode	Reg
xxx.W	111	000
xxx.L	111	001
d16(PC)	111	010
d8(PC,Xi)	111	011
#<data>	111	100
Erläuterung siehe Kapitel 5		

Siehe auch: CMP, CMPI, CMPM

Vergleiche Konstante
compare immediate

CMPI

Ziel - Konstante

Operandgröße: CMPI.B Byte (8 Bit)
CMPI.W Wort (16 Bit)
CMPI.L Langwort (32 Bit)

Assembler Syntax: CMPI.x #<data>, <ea> (Quelle, Ziel)
(x entspricht B, Wr L)

Beschreibung:

Die Konstante, die im Speicher unmittelbar dem Befehlswort folgt, wird binär vom Zieloperand <ea> subtrahiert, und das Condition Code Register wird entsprechend gesetzt.

Der Zieloperand wird nicht geändert.

Die Größe der Konstante entspricht der Operandgröße.

X	N	Z	V	C
-	*	*	*	*

Condition Code Register:

C wird gesetzt, wenn ein "Leihen" generiert wird.
Wird sonst zurückgesetzt.
V wird gesetzt, wenn ein Überlauf generiert wird.
Wird sonst zurückgesetzt.
Z wird gesetzt, wenn das Ergebnis gleich Null ist.
Wird sonst zurückgesetzt.
N wird gesetzt, wenn das Ergebnis negativ ist.
Wird sonst zurückgesetzt.
X bleibt unverändert.

CMPI

Vergleiche Konstante
compare immediate

Format des Befehlswortes:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	0	Größe		Effektive Adresse Mode			Register		
1. Argumentwort: Wort Daten								bzw. Byte Daten							
2. Argumentwort: Langwort-Daten (einschließlich voriges Wort)															

Bit 7..6 Größe-Feld:
00 Byte-Befehl CMPI.B
01 Wort-Befehl CMPI.W
10 Langwort-Befehl CMPI.L

Aufbau der Argumentwörter siehe Kap. 3.8

Bit 5..0 Die Effektive Adresse wählt den Zieloperand an. Die folgenden Adressierungsarten sind erlaubt:

Adr.-Art	Mode	Reg
Dn	000	R:Dn
An	nicht erlaubt	
(An)	010	R:An
(An)+	011	R:An
-(An)	100	R:An
d16(An)	101	R:An
d8(An,Xi)	110	R:An

Adr.-Art	Mode	Reg
xxx.W	111	000
xxx.L	111	001
d16(PC)	nicht erlaubt	
d8(PC,Xi)	nicht erlaubt	
#<data>	nicht erlaubt	
Erläuterung siehe Kapitel 5		

Argumentwort:
Bit 7..6 = 00 -> Datenfeld ist die niederwertige Hälfte des 1. Argumentwortes
Bit 7..6 = 01 -> Datenfeld ist das 1. Argumentwort
Bit 7..6 = 10 -> Datenfeld ist 1. + 2. Argumentwort

Siehe auch: CMP, CMPA, CMPM

Vergleiche Speicherinhalt
compare memory

CMPM

Ziel - Quelle

Operandgröße: CMPM.B Byte (8 Bit)
CMPM.W Wort (16 Bit)
CMPM.L Langwort (32 Bit)

Assembler Syntax: CMPM.x (An)+, (An)+ (Quelle, Ziel)
(x entspricht B, W, L)

Beschreibung:

Der Quelloperand wird binär vom Zieloperand subtrahiert, und das Condition Code Register wird entsprechend gesetzt.

Die Operanden werden mit der Adressregister indirekter Adressierung adressiert, siehe Kap. 5.1.5

Der Zieloperand wird nicht geändert.
Die Größe der Konstante entspricht der Operandgröße.

Condition Code Register:

X	N	Z	V	C
-	*	*	*	*

C wird gesetzt, wenn ein "Leihen" generiert wird.
Wird sonst zurückgesetzt.
V wird gesetzt, wenn ein Überlauf generiert wird.
Wird sonst zurückgesetzt.
Z wird gesetzt, wenn das Ergebnis gleich Null ist.
Wird sonst zurückgesetzt.
N wird gesetzt, wenn das Ergebnis negativ ist.
Wird sonst zurückgesetzt.
X bleibt unverändert.

CMPM

Vergleiche Speicherinhalt
compare memory

Format des Befehlswortes:

15	14	13	12	11 10 9	8	7 6	5	4	3	2 1 0
1	0	1	1	Ziel-Register	1	Größe	0	0	1	Quell-Register

Bit 11..9 wählt einer der acht Adressregister als Zielregister an.

Bit 7..6 Größe-Feld:
00 Byte-Befehl CMPM.B
01 Wort-Befehl CMPM.W
10 Langwort-Befehl CMPM.L

Aufbau der Argumentwörter siehe Kap. 3.8

Bit 2..0 wählt einer der acht Adressregister als Quellegister an.

Siehe auch: CMP, CMPA, CMPM

prüfe, dekrementiere und springe
test condition, decrement and branch

DBcc

DBcc ist ein überbegriff,
siehe Befehlsliste

Wenn (Bedingung = unwahr), dann
(Dn - 1 -> Dn; wenn Dn ≠ -1 dann PC + d -> PC)

Größe der Adressdifferenz: Wort (16 Bit)

Assembler Syntax: DBcc Dn, Marke

Beschreibung:

Die Bezeichnung DBcc im Kopf dieser Seite ist stellvertretend für die Befehle DBCC, DBCS, DBEQ, DBGE, DBGT, DBHI, DBLE, DBLS, DBLT, DEMI, DBNE, DBPL, DBVC, DBVS, DBF und DBT. Wir fassen alle diese Befehle hier zusammen.

Die Befehle DBcc sind gedacht für die Programmierung von Schleifen.

Bei den Befehlen werden die Bits N (Negative), Z (Zero), V (oVerflow) und C (Carry) des Condition Code Registers benutzt.

Die Bedingungen, die geprüft werden, sind ähnlich den Bedingungen für bedingte Sprünge - siehe bei Bcc.
Wenn die zu prüfende Bedingung wahr ist, passiert gar nichts, die Schleife ist damit beendet. Der nächste Befehl ist dann der, der im Speicher direkt auf den DBcc-Befehl folgt.

DBcc

prüfe, dekrementiere und springe
test condition, decrement and branch

Aber wenn die Bedingung unwahr ist, tritt der DBcc-Befehl in Aktion. Es passiert dann folgendes:

> Das **niederwertige** Wort des angewählten Datenregisters (das sich also in Bit 15..0 befindet) wird um eins (1) vermindert.
> Wenn es bei der Verminderung nicht den Wert -1 erhält, findet ein Sprung zu der spezifizierten Marke statt. Das Datenregister wird als Schleifenzähler (loop counter) bezeichnet.
> Wenn das niederwertige Wort des Datenregister also am Anfang den Wert -1 hatte, wird die Schleife 65536 mal durchlaufen (angenommen, daß die Bedingung nicht vorher erfüllt wird).

BEISPIEL

Dieses Programm soll untersuchen, ob in dem String (Zeichenkette), der sich zwischen BUF und BUFE befindet, das Zeichen "w" enthalten ist.

Wenn ja, dann soll die Adresse des ersten "w"s ermittelt werden.

```
      MOVE.W  BUF, A5       ; Anfang Buffer zu A5
      MOVE.W  BUFE-BUF, D5  ; Laenge Buffer zu D5
LOOP: CMPI.B  "w", (A5)+    ; Ist dort ein "w" ?
      DBNE    D5, LOOP      ; Wenn nein, suche weiter
      TST.W   D5            ; Wenn D5 negativ, dann
      BMI     NOTFOUND      ;   kein "w" gefunden
      JUMP    FOUND         ; Gefunden, Adresse in A5

BUF   DC.B    "So ein Tag, so wunderschoen wie heute"
BUFE
```

prüfe, dekrementiere und springe
test condition, decrement and branch **DBcc**

Bei den Befehlen haben wir neben den deutschen auch die originalen (amerikanischen) Befehlsbezeichnungen gegeben. Damit können Sie sich die Mnemonics besser merken.

DBCC beende Schleife, wenn C-Bit zurückgesetzt ist
Terminate if Carry is Clear

DBCS beende Schleife, wenn C-Bit gesetzt ist
Terminate if Carry is Set

DBEQ beende Schleife, wenn gleich.
Terminate if EQual
Die Schleife wird beendet, wenn das Z-Bit (Zero) gesetzt ist.

DBGE beende Schleife, wenn größer oder gleich.
Terminate on Greater than or Egual
Die Schleife wird beendet, wenn das N-Bit (Negative) und das V-Bit (oVerflow) entweder beide gesetzt oder beide zurückgesetzt sind.
DBGE ist für Binärzahlen mit Vorzeichen gedacht.

DBGT beende Schleife, wenn größer
Terminate on Greater Than
Die Schleife wird beendet, wenn
o das N-Bit und das V-Bit gesetzt sind und das Z-Bit zurückgesetzt ist,
oder
o das N-Bit, das V-Bit und das Z-Bit alle zurückgesetzt sind.
DBGT ist für Binärzahlen mit Vorzeichen gedacht, ist sonst ähnlich DBHI.

DBcc

prüfe, dekrementiere und springe
test condition, decrement and branch

DBHI beende Schleife, wenn höher
Terminate on Higher than
Die Schleife wird beendet, wenn das C-Bit und das Z-Bit beide zurückgesetzt sind. DBGE ist für Binärzahlen ohne Vorzeichen gedacht, ist. sonst ähnlich DBGT.

DBLE beende Schleife, wenn kleiner oder gleich
Terminate on Less than or Equal
Die Schleife wird beendet, wenn
- o das Z-Bit gesetzt ist,
  oder
- o das N-Bit gesetzt und das V-Bit zurückgesetzt ist,
  oder
- o das N-Bit zurückgesetzt und das V-Bit gesetzt ist.

DBLE ist für Binärzahlen mit Vorzeichen gedacht, ist sonst ähnlich DBLS.

DBLS beende Schleife, wenn niedriger oder gleich
Terminate on Lower or Same
Die Schleife wird beendet, wenn das C-Bit, das Z-Bit oder beide gesetzt sind.
DBLS ist für Binärzahlen ohne Vorzeichen gedacht, ist sonst ähnlich DBLE.

prüfe, dekrementiere und springe
test condition, decrement and branch

DBcc

DBLT beende Schleife, wenn kleiner
Terminate on Less Than
Die Schleife wird beendet, wenn
- o das N-Bit gesetzt und das V-Bit zurückgesetzt,

oder
- o das N-Bit zurückgesetzt und das V-Bit gesetzt ist.

DBLT ist für Binärzahlen mit Vorzeichen gedacht.

DBMI beende Schleife, wenn Minus
Terminate on Minus
Die Schleife wird beendet, wenn das N-Bit gesetzt ist.
DBMI ist für Binärzahlen mit Vorzeichen gedacht.

DBNE beende Schleife, wenn ungleich
Terminate on Not Equal
Die Schleife wird beendet, wenn das Z-Bit zurückgesetzt ist.

DBPL beende Schleife, wenn Plus
Terminate on PLus
Die Schleife wird beendet, wenn das N-Bit zurückgesetzt ist.
DBPL ist für Binärzahlen mit Vorzeichen gedacht.

DBVC beende Schleife, wenn kein Überlauf
Terminate on oVerflow Clear
Die Schleife wird beendet, wenn das V-Bit zurückgesetzt ist.

DBcc

prüfe, dekrementiere und springe
test condition, decrement and branch

```
DBVS beende Schleife, wenn Überlauf
     Terminate on oVerflow Set
     Die Schleife wird beendet, wenn das V-Bit
     gesetzt ist.

DBF  beende Schleife nie
     never terminate
     Die Schleife wird nur durch den Zähler beendet.
     Viele Assembler akzeptieren DBRA als synonym für
     DBF

DBT  beende Schleife immer
     always terminate
     Dieser Befehl bildet überhaupt keine Schleife.
```

X	N	Z	V	C
-	-	-	-	-

Condition Code Register:
bleibt unverändert

prüfe, dekrementiere und springe
test condition, decrement and branch

DBcc

Format des Befehlswortes:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	Bedingung				1	1	0	0	1	Daten-Register		
16-Bit Adressdifferenz bis zum Anfang der Schleife															

Bit 11..8 Bedingungsfeld

Bedingung	Befehl
0000	DBT
0001	DBF,DBRA
0010	DBHI
0011	DBLS
0100	DBCC
0101	DBCS
0110	DBNE
0111	DBEQ

Bedingung	Befehl
1000	DBVC
1001	DBVS
1010	DBPL
1011	DEMI
1100	DBGE
1101	DBLT
1110	DBGT
1111	DBLE

Bit 2..0 Feld Datenregister Dn
wählt das Datenregister an, das als Schleifenzähler benutzt wird.

Feld Adress-differenz:
wird als Binärzahl mit Vorzeichen aufgefaßt. Die Sprungadresse ist (PC + Adress-Differenz).

DIVS

Dividiere mit Vorzeichen
signed divide

Ziel / Quelle -> Ziel (32/16 -> 16r:16q)
Divident / Divisor -> Quotient, Rest

Operandgröße:

o Eingangswerte: Divisor: Wort (16 Bit)
Divident: Langwort (32 Bit)

o Ergebniswerte: Quotient: Wort (16 Bit)
Rest: Wort (16 Bit)

Assembler Syntax: DIVS <ea>, Dn (Quelle, Ziel)

Beschreibung:
Der Zieloperand wird durch den Quelloperand dividiert.

Die beiden Operanden werden als Binärzahl mit Vorzeichen aufgefaßt.

Der Zieloperand ist ein Langwort (32 Bit), der Quelloperand ist ein Wort (16 Bit).

Das Ergebnis der Division wird im Zieloperand abgespeichert. Der Quotient kommt dabei in die niederwertige Hälfte (Bit 15..0) des Zieloperanden, der Rest der Teilung kommt in die hochwertige Hälfte (Bit 31..16) des Zieloperanden.

Der Rest erhält das gleiche Vorzeichen wie der Divident.

Dividiere mit Vorzeichen
signed divide

DIVS

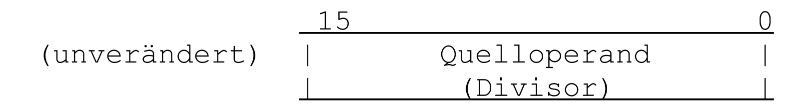
Wenn der Quotient größer ist als ein 16-Bit Wort mit Vorzeichen, tritt ein Überlauf auf. In dem Fall ändern sich nur die Werte des Condition Code Registers, die Operanden bleiben unverändert.

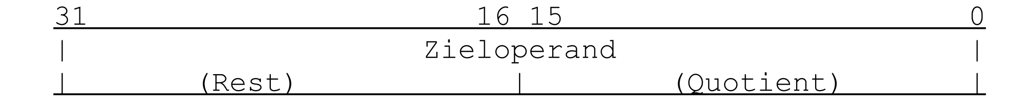
Vor der Operation:

```
                                  15                       0
                                  |      Quelloperand      |
                                  |        (Divisor)       |

31                          16 15                          0
|                        Zieloperand                       |
|                        (Divident)                        |
```

Nach der Operation: Ziel / Quelle -> Ziel





DIVS

Dividiere mit Vorzeichen
signed divide

Bei der Operation können zwei Sonderfälle auftreten:

1. Division durch Null. Es wird eine Exception 5 ausgelöst.

2. Eine große Zahl wird durch eine kleine Zahl dividiert, und das Ergebnis paßt nicht in 16 Bit. Das Ergebnis ist eine Überlauf. In dem Fall ändern sich nur die Werte des Condition Code Registers, die Operanden bleiben unverändert.

Condition Code Register:

X	N	Z	V	C
-	*	*	*	0

C wird zurückgesetzt.
V wird gesetzt, wenn ein Überlauf aufgetreten ist. Wird sonst zurückgesetzt.
Z wird gesetzt, wenn der Quotient gleich Null ist. Z ist nicht definiert bei Überlauf oder Division durch Null. Wird sonst zurückgesetzt.
N wird gesetzt, wenn das Ergebnis negativ ist. N ist nicht definiert bei Überlauf oder Division durch Null. Wird sonst zurückgesetzt.
X bleibt unverändert.

Dividiere mit Vorzeichen
signed divide

DIVS

Format des Befehlswortes:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	Daten-Register				1	1	1	Effektive Adresse Mode			Effektive Adresse Register	

Bit 11..9 Registerfeld: wählt eines der acht Datenregister als Zieloperand an.

Bit 5..0 Wählt die Effektive Adresse des Quelloperanden an. Die folgenden Adressierungsarten sind erlaubt:

Adr.-Art	Mode	Reg
Dn	000	R:Dn
An	nicht erlaubt	
(An)	010	R:An
(An)+	011	R:An
-(An)	100	R:An
d16(An)	101	R:An
d8(An,Xi)	110	R:An

Adr.-Art	Mode	Reg
xxx.W	111	000
xxx.L	111	001
d16(PC)	111	010
d8(PC,Xi)	111	011
#<data>	111	100
Erläuterung siehe Kapitel 5		

Benutzen Sie
DIVU: wenn das Vorzeichen bei der Division nicht berücksichtigt werden soll.

Siehe auch: MULS, MULU



Dividiere ohne Vorzeichen
divide unsigned

Ziel / Quelle -> Ziel (32/16 -> 16r:16q)
Divident / Divisor -> Quotient, Rest

Operandgröße:

o Eingangswerte Divisor: Wort (16 Bit)
Divident: Langwort (32 Bit)

o Ergebniswerte: Quotient: Wort (16 Bit)
Rest: Wort (16 Bit)

Assembler Syntax: DIVU <ea>, Dn (Quelle, Ziel)

Beschreibung:
Der Zieloperand wird durch den Quelloperand dividiert.

Die beiden Operanden werden als Binärzahl ohne Vorzeichen aufgefaßt.

Der Zieloperand ist ein Langwort (32 Bit), der Quelloperand ist ein Wort (16 Bit).

Das Ergebnis der Division wird im Zieloperand abgespeichert. Der Quotient kommt dabei in die niederwertige Hälfte (Bit 15..0) des Zieloperanden. Der Rest der Teilung kommt in die hochwertige Hälfte (Bit 31..16) des Zieloperanden.

Wenn der Quotient größer ist als ein 16-Bit Wort mit Vorzeichen, tritt ein Überlauf auf. In dem Fall ändern sich nur die Werte des Condition Code Registers, die Operanden bleiben unverändert.

Dividiere ohne Vorzeichen
divide unsigned

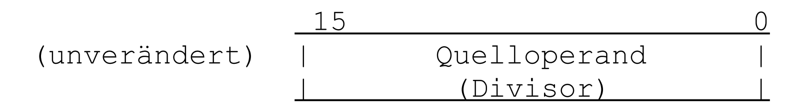
DIVU

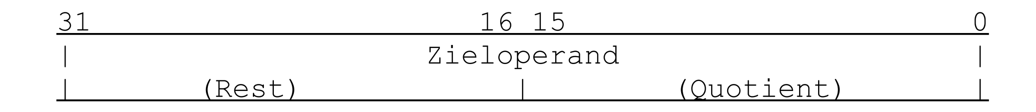
Vor der Operation:

```
                          15                          0
                         ______________________________
                         |        Quelloperand        |
                         |_________(Divisor)__________|

31                      16 15                          0
________________________________________________________
|                    Zieloperand                       |
|____________________(Divident)________________________|
```

Nach der Operation: Ziel / Quelle -> Ziel





DIVU

Dividiere ohne Vorzeichen
divide unsigned

Bei der Operation können zwei Sonderfälle auftreten:

1. Division durch Null. Es wird eine Exception 5 ausgelöst.

2. Eine große Zahl wird durch eine kleine Zahl dividiert, und das Ergebnis paßt nicht in 16 Bit. Das Ergebnis ist ein Überlauf. In dem Fall ändern sich nur die Werte des Condition Code Registers, die Operanden.bleiben unverändert.

Condition Code Register:

X	N	Z	V	C
-	*	*	*	0

C wird zurückgesetzt.
V wird gesetzt, wenn ein Überlauf aufgetreten ist. Wird sonst zurückgesetzt.
Z wird gesetzt, wenn der Quotient gleich Null ist. Z ist nicht definiert bei Überlauf oder Division durch Null. Wird sonst zurückgesetzt.
N wird gesetzt, wenn das Ergebnis negativ ist. N ist nicht definiert bei Überlauf oder Division durch Null. Wird sonst zurückgesetzt.
X bleibt unverändert.

Dividiere ohne Vorzeichen
divide unsigned

DIVU

Format des Befehlswortes:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	Daten-Register			0	1	1	Effektive Adresse Mode			Register		

Bit 11..9 Registerfeld: wählt eines der acht Datenregister als Zieloperand an.

Bit 5..0 Wählt die Effektive Adresse des Quelloperanden an. Die folgenden Adressierungsarten sind erlaubt:

Adr.-Art	Mode	Reg
Dn	000	R:Dn
An	nicht erlaubt	
(An)	010	R:An
(An)+	011	R:An
-(An)	100	R:An
d16(An)	101	R:An
d8(An,Xi)	110	R:An

Adr.-Art	Mode	Reg
xxx.W	111	000
xxx.L	111	001
d16(PC)	111	010
d8(PC,Xi)	111	011
#<data>	111	100
Erläuterung siehe Kapitel 5		

Benutzen Sie

DIVS: wenn das Vorzeichen bei der Division berücksichtigt werden soll.

Siehe auch: MULS, MULU

EOR

Exclusives ODER
exclusive or logical

Quelle ≠ Ziel -> Ziel

Operandgröße: EOR.B Byte (8 Bit)
EOR.W Wort (16 Bit)
EOR.L Langwort (32 Bit)

Assembler Syntax:
EOR.x Dn, <ea>
(x entspricht B, W, L)

Operation:
Dn ≠ <ea> -> <ea>

Beschreibung :

Der Quelloperand wird mit dem Zieloperand bitweise EXCLUSIVE-ODER-verknüpft , und das Ergebnis wird im Zieloperand abgespeichert.

Zur Erinnerung die EXCLUSIVE-ODER-Verknüpfungen:

0 ≠ 0 = 0
0 ≠ 1 = 1
1 ≠ 0 = 1
1 ≠ 1 = 0

Das Ergebnisbit wird gesetzt, wenn die Eingangsbits ungleich sind.

Einer der beiden Operanden muß ein Datenregister sein.

Die Größe des Operanden sowie die Angabe, welcher Operand das Datenregister ist, sind im Mode-Feld enthalten.

Exclusives ODER
exclusive or logical

EOR

```
  X    N    Z    V    C
 ___________________________
|    |    |    |    |    |
| -  | *  | *  | 0  | 0  |
|____|____|____|____|____|
```

Condition Code Register:

C wird zurückgesetzt.
V wird zurückgesetzt.
Z wird gesetzt, wenn das Ergebnis gleich Null ist.
   Wird sonst zurückgesetzt.
N wird gesetzt, wenn das hochwertige Bit des
   Ergebnisses gesetzt ist (zeigt ein negatives
   Ergebnis). Wird sonst zurückgesetzt.
X bleibt unverändert.

Format des Befehlswortes:

```
| 15  14  13  12| 11  10  9   8 | 7   6   5   4 | 3   2   1   0 |
|   |   |   |   |            |            |                       |
| 1 | 0 | 1 | 1 |  Daten-    |Operations-|    Effektive Adresse  |
|___|___|___|___| REgister   |   Mode     |  Mode      | Register |
```

Bit 11..9 Registerfeld: wählt eines der acht
                        Datenregister an.

Bit 8..6 Feld Operationsmode:

```
EOR.B EOR.W EOR.L   Operation
 100   101   110    Dn  ^ <ea> -> <ea>
```

EOR

Exclusives ODER
exclusive or logical

Bit 5..0 spezifiziert die Effektive Adresse des Zieloperanden. Die folgenden Adressierungsarten sind erlaubt:

Adr.-Art	Mode	Reg
Dn	000	R:Dn
An	nicht erlaubt	
(An)	010	R:An
(An)+	011	R:An
-(An)	100	R:An
d16(An)	101	R:An
d8(An,Xi)	110	R:An

Adr.-Art	Mode	Reg
xxx.W	111	000
xxx.L	111	001
d16(PC)	nicht erlaubt	
d8(PC,Xi)	nicht erlaubt	
#<data>	nicht erlaubt	
Erläuterung siehe Kapitel 5		

Benutzen Sie
EORI: wenn einer der Operanden eine Konstante ist.

Siehe auch: AND, OR, NOT, TST

Exclusives ODER mit Konstante
exclusive or immediate

EORI

Konstante ≠ Ziel -> Ziel

Operandgröße: EORI.B Byte (8 Bit)
EORI.W Wort (16 Bit)
EORI.L Langwort (32 Bit)

Assembler Syntax: EORI.x #<data>, <ea> (Quelle, Ziel)
(x entspricht B, W, L)

Beschreibung:

Die Konstante, die im Speicher unmittelbar dem Befehlswort folgt, wird mit dem Zieloperand <ea> bitweise EXCLUSIVE-ODER-verknüpft. Das Ergebnis wird im Zieloperand <ea> abgespeichert.

Die Größe der Konstante entspricht der Operandgröße.

Condition Code Register:

X	N	Z	V	C
-	*	*	0	0

C wird zurückgesetzt.
V wird zurückgesetzt.
Z wird gesetzt, wenn das Ergebnis gleich Null ist. Wird sonst zurückgesetzt.
N wird gesetzt, wenn das hochwertige Bit des Ergebnisses gesetzt ist (zeigt ein negatives Ergebnis). Wird sonst zurückgesetzt.
X bleibt unverändert.

EORI

Exclusives ODER mit Konstante
exclusive or immediate

Format des Befehlswortes:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	0	Größe		Effektive Adresse Mode			Register		
1. Argumentwort: Wort Daten								bzw. Byte Daten							
2. Argumentwort: Langwort Daten (einschließlich voriges Wort)															

Bit 7..6 Größe-Feld:
00 Byte-Befehl EORI.B
01 Wort-Befehl EORI.W
10 Langwort-Befehl EORI.L

Aufbau der Argumentwörter siehe Kap. 3.8

Bit 5..0 Die Effektive Adresse wählt den Zieloperand an. Die folgenden Adressierungsarten sind erlaubt:

Adr.-Art	Mode	Reg
Dn	000	R:Dn
An	nicht erlaubt	
(An)	010	R:An
(An)+	011	R:An
-(An)	100	R:An
d16(An)	101	R:An
d8(An,Xi)	110	R:An

Adr.-Art	Mode	Reg
xxx.W	111	000
xxx.L	111	001
d16(PC)	nicht erlaubt	
d8(PC,Xi)	nicht erlaubt	
#<data>	nicht erlaubt	
Erläuterung siehe Kapitel 5		

Argumentwort:
Bit 7..6 = 00 -> Datenfeld ist die niederwertige Hälfte des 1. Argumentwortes
Bit 7..6 = 01 -> Datenfeld ist das 1. Argumentwort
Bit 7..6 = 10 -> Datenfeld ist 1. +2. Argumentwort

Siehe auch: ANDI, ORI, EOR, NOT, TST

Exclusives ODER mit Konstante
exclusive or immediate

EORI zum CCR

Konstante ≠ CCR -> CCR

Operandgröße: Byte (8 Bit)

Assembler Syntax: EORI #<data>, CCR (Quelle, Ziel)

Beschreibung:

Die im Speicher unmittelbar dem Befehlswort folgende Konstante wird mit dem Condition Code Register bitweise Exclusive-ODER-verknüpft. Das Ergebnis wird im Condition Code Register abgespeichert.

Die Operandgröße ist ein Byte.

Condition Code Register:

X	N	Z	V	C
*	*	*	*	*

C ändert sich, wenn Bit 0 der Konstante gesetzt ist. Bleibt sonst unverändert.
V ändert sich, wenn Bit 1 der Konstante gesetzt ist. Bleibt sonst unverändert.
Z ändert sich, wenn Bit 2 der Konstante gesetzt ist. Bleibt sonst unverändert.
N ändert sich, wenn Bit 3 der Konstante gesetzt ist. Bleibt sonst unverändert.
X ändert sich, wenn Bit 4 der Konstante gesetzt ist. Bleibt sonst unverändert.

EORI zum CCR

Exclusives ODER mit Konstante
exclusive or immediate

Format des Befehlswortes:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	0	0	0	1	1	1	1	0	0
0	0	0	0	0	0	0	0	Konstante (8 Bit)							

Benutzen Sie

EORI zum SR: wenn Sie auch das System Byte Exclusiv-ODER-verknüpfen möchten.

ANDI zum CCR: wenn Sie das CCR UND-verknüpfen möchten.

ORI zum CCR: wenn Sie das CCR ODER-verknüpfen möchten.

MOVE zum CCR: wenn Sie das CCR ohne Rücksicht auf die bestehenden Bits ändern möchten.

Siehe auch: ANDI zum SR, ORI zum SR

Exclusives ODER mit Konstante
exclusive or immediate
(privilegierter Befehl)

EORI zum CCR

Wenn Supervisor Mode: Konstante ≠ SR -> SR

Wenn User Mode: **Auslösung Exception 8 (Kap 6)**
(Verletzung Privilegium)

Operandgröße: Wort (16 Bit)

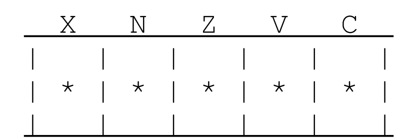
Assembler Syntax: ORI f<data>, SR (Quelle, Ziel)

Beschreibung:

Wenn der Prozessor sich im Supervisor Mode befindet, wird die Konstante, die im Speicher unmittelbar dem Befehlswort folgt, mit dem Status Register bitweise "Exclusive ODER" verknüpft. Das Ergebnis wird im Status Register abgespeichert.

Wenn der Prozessor dagegen im User Mode ist, wird eine Exception ausgelöst.

Das Status Register wird in Kap. 2 beschrieben. Die Operandgröße ist ein Wort (16 Bit).



Exclusives ODER mit Konstante
exclusive or immediate
(privilegierter Befehl)

Condition Code Register:

X	N	Z	V	C
*	*	*	*	*

C ändert sich, wenn Bit 0 der Konstante gesetzt ist. Bleibt sonst unverändert.
V ändert sich, wenn Bit 1 der Konstante gesetzt ist. Bleibt sonst unverändert.
Z ändert sich, wenn Bit 2 der Konstante gesetzt ist. Bleibt sonst unverändert.
N ändert sich, wenn Bit 3 der Konstante gesetzt ist. Bleibt sonst unverändert.
X ändert sich, wenn Bit 4 der Konstante gesetzt ist. Bleibt sonst unverändert.

Format des Befehlswortes:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	0	0	1	1	1	1	1	0	0
K o n s t a n t e (1 6 Bit)															

Benutzen Sie

ANDI zum SR:	wenn Sie das SR UND-verknüpfen möchten.
ORI zum SR:	wenn Sie das SR ODER-verknüpfen möchten.
MOVE zum SR:	wenn Sie das SR ohne Rücksicht auf die bestehenden Bits ändern möchten.
EORI zum CCR	wenn Sie nur das Condition Code Register ändern möchten.

Siehe auch: EORI zum CCR, ORI zum SR, ANDI zum SR

Vertausche Register
exchange registers

EXG

Quelle <-> Ziel
Operandgröße: Langwort (32 Bit)

Assembler Syntax:

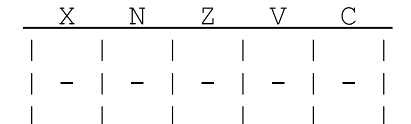
```
EXG  Dn, Dn  (Quelle, Ziel)
EXG  An, An
EXG  Dn, An
```

Beschreibung:

Die Inhalte zweier Register werden vertauscht.
Es werden alle 32 Bits der Register getauscht.

Es gibt drei verschiedene Moden:

```
EXG Dn, Dn  vertausche Datenregister
EXG An, An  vertausche Adressregister
EXG Dn, An  vertausche ein Datenregister und ein
            Adressregister.
```



X	N	Z	V	C
-	-	-	-	-

Condition Code Register:
keine Änderung

EXG

Vertausche Register
exchange registers

Format des Befehlswortes:

15	14	13	12	11 10 9	8	7 6 5 4 3	2 1 0
1	1	0	0	Ziel-Register	1	Operations-Mode	Quell-Register

```
Bit 11..9   Registerfeld: wählt eines der Daten- oder
            Adressregister als Zieloperand an.
            Operationsmode:  Anwahl von:
              01000        Datenregister
              01001        Adressregister
              10001        Datenregister


Bit 7..3    Wählt den Operationsmode an.
              01000 vertausche Datenregister
              01001 vertausche Adressregister
              10001 vertausche ein Datenregister und
                    ein Adressregister.


Bit 2..0    Registerfeld: wählt eines der Daten- oder
            Adressregister als Zieloperand an.
            Operationsmode:  Anwahl von:
              01000         Datenregister
              01001         Adressregister
              10001         Adressregister
```

```
Benutzen Sie
MOVE oder MOVEA:    wenn der Datentransfer nur in
                    einer Richtung stattfinden soll
```

Erweitere Vorzeichen
extend sign

EXT

Ziel mit erweitertem Vorzeichen -> Ziel

Operandgröße: EXT.W Wort (16 Bit)
EXT.L Langwort (32 Bit)

Assembler Syntax: EXT.x Dn
(x entspricht W, L)

Beschreibung:

Das Vorzeichen der Daten im angewählten Datenregister wird erweitert.

Bei dem Befehl EXT.W wird das Vorzeichen von einem Byte zu einem Wort erweitert. Dazu wird das Vorzeichen des Bytes, das sich in Bit 7 befindet, in den Bits 8..15 hineinkopiert.

Bei dem Befehl EXT.L wird das Vorzeichen von einem Wort zu einem Langwort erweitert. Dazu wird das Vorzeichen des Wortes, das sich in Bit 15 befindet, in den Bits 16..31 hineinkopiert.

Der Operand kann nur ein Datenregister sein.
Mehr Information über Daten mit oder ohne Vorzeichen in Kap. 3.

EXT

Erweitere Vorzeichen
extend sign

Condition Code Register:

X	N	Z	V	C
-	*	*	0	0

C wird zurückgesetzt.
V wird zurückgesetzt.
Z wird gesetzt, wenn das Ergebnis gleich Null ist.
  Wird sonst zurückgesetzt.
N wird gesetzt, wenn das Ergebnis negativ ist.
  Wird sonst zurückgesetzt.
X bleibt unverändert.

Format des Befehlswortes:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	Operations-Mode			0	0	0	Daten-Register		

Bit 8..6 Feld Operations-Mode:

010 EXT.W - Wort-Befehl. Das Vorzeichen wird von einem Byte zu einem Wort erweitert.

011 EXT.L - Langwort-Befehl. Das Vorzeichen wird von einem Wort zu einem Langwort erweitert.

Bit 2..0 Register-Feld
Wählt das Datenregister Dn an, dessen Vorzeichenbit erweitert werden soll.

Siehe auch: NEG, NEGX, NBCD

Löse Illegal-Exception aus
take illegal instruction-trap

ILLEGAL

Die Exception 4 wird ausgelöst (siehe Kap. 6).

Operandgröße: keine

Assembler Syntax: ILLEGAL

Beschreibung:

Der Prozessor gelangt in den Supervisor Mode über die Exception 4, siehe Kap. 6

Auch andere, nicht erlaubte Befehle lösen eine Exception 4 aus. Aus Gründen der Übersichtlichkeit und Kompatibilität wird dieser Befehl bevorzugt über andere, nicht erlaubte Befehle.

X	N	Z	V	C
-	-	-	-	-

Condition Code Register:
bleibt unverändert

Format des Befehlswortes:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	1	0	1	1	1	1	1	1	0	0

($4AFC)

Siehe auch: TRAP

JMP

Springe unbedingt
jump

Zieladresse -> Programmzähler

Operandgröße: keine

Assembler Syntax: JMP <ea>

Beschreibung:

Der Programmablauf geht bei der angebenen effektiven Adresse weiter.

X	N	Z	V	C
-	-	-	-	-

Condition Code Register:
bleibt unverändert

Format des Befehlswortes:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	1	1	Effektive Adresse					
										Mode			Register		

Springe unbedingt
jump

JMP

Bit 5..0 spezifiziert die nächste Befehlsadresse
Die folgenden Adressierungsarten sind erlaubt:

Adr.-Art	Mode	Reg
Dn	nicht erlaubt	
An	nicht erlaubt	
(An)	010	R:An
(An)+	nicht erlaubt	
-(An)	nicht erlaubt	
d16(An)	101	R:An
d8(An,Xi)	110	R: An

Adr.-Art	Mode	Reg
xxx.W	111	000
xxx.L	111	001
d16(PC)	111	010
d8(PC,Xi)	111	011
#<data>	nicht erlaubt	
Erläuterung siehe Kapitel 5		

PROGRAMMIERHINWEIS:
Die Sprungadresse kann auch das Ergebnis einer Berechnung sein. Der Sprung kann dann direkt zur richtigen Stelle innerhalb einer Tabelle stattfinden.

So richtet sich beim Befehl JMP 8(A3, D4) die Sprungadresse nach der Summe zweier Register und einer Konstante.

Benutzen Sie

JSR: wenn Sie ein Unterprogramm aufrufen möchten.

BRA: wenn Sie relativ adressieren und dabei Speicherplatz sparen möchten. (BRA finden Sie unter dem Oberbegriff Bcc).

JSR

Aufruf Unterprogramm
jump to subroutine

SP - 4 -> SP; PC -> (SP); <ea> -> PC

Assembler Syntax: JSR <ea>

Beschreibung:

Die 32-Bit Adresse des Befehls, der im Speicher direkt auf den BSR-Befehl folgt, wird auf den Stack gepushed.

Danach geht die Ausführung des Programms weiter bei der angegebenen Speicheradresse.

Ein Beispiel wird in Kap. 3.10 gegeben.

PROGRAMMIERHINWEIS:

Beenden Sie das Unterprogramm mit RTS.
Die Programmausfuhr geht dann weiter mit dem Befehl, der im Speicher direkt auf den JSR-Befehl folgt.

X	N	Z	V	C
-	-	-	-	-

Condition Code Register:
bleibt unverändert

Aufruf Unterprogramm
jump to subroutine

JSR

Format des Befehlswortes:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	1	0	Effektive Adresse					
										Mode			Register		

Bit 5..0 Die Effektive Adresse wählt die Sprung-dresse an. Die folgenden Adressierungsarten sind erlaubt:

Adr.-Art	Mode	Reg
Dn	nicht erlaubt	
An	nicht erlaubt	
(An)	010	R:An
(An)+	nicht erlaubt	
-(An)	nicht erlaubt	
d16(An)	101	R:An
d8(An,Xi)	110	R:An

Adr.-Art	Mode	Reg
xxx.W	111	000
xxx.L	111	001
d16(PC)	111	010
d8(PC,Xi)	111	011
#<data>	nicht erlaubt	
Erläuterung siehe Kapitel 5		

Benutzen Sie

BSR: wenn Sie die Sprungadresse relativ adressieren und dabei Speicherplatz sparen möchten. (BRA finden Sie unter dem Über-begriff Bcc).

JMP oder BRA: wenn keine Rückkehr erwünscht ist.

LEA

Lade effektive Adresse im Register
load effective address

Quelle -> Ziel

Operandgröße: Langwort (32 Bit)

Assembler Syntax: LEA <ea>, An (Quelle, Ziel)

Beschreibung:
Der Quelloperand wird in das spezifizierte Adressregister kopiert.

Es werden immer alle 32 Bits des Adressregisters beeinflußt.

X	N	Z	V	C
-	-	-	-	-

Condition Code Register:
bleibt unverändert

Lade effektive Adresse im Register
load effective address

LEA

Format des Befehlswortes:

15	14	13	12	11 10 9	8	7	6	5 4 3	2 1 0
0	1	0	0	Register An	1	1	1	Effektive Adresse Mode	Register

Bit 11..9 Registerfeld: Wählt eines der acht Adressregister als Zieloperand an.

Bit 5..0 Wählt die Effektive Adresse des Quelloperanden an. Die folgenden Adressierungsarten sind erlaubt:

Adr.-Art	Mode	Reg
Dn	nicht erlaubt	
An	nicht erlaubt	
(An)	010	R:An
(An)+	nicht erlaubt	
-(An)	nicht erlaubt	
d16(An)	101	R:An
d8(An,Xi)	110	R:An

Adr.-Art	Mode	Reg
xxx.W	111	000
xxx.L	111	001
d16(PC)	111	010
d8(PC,Xi)	111	011
#<data>	nicht erlaubt	
Erläuterung siehe Kapitel 5		

Benutzen Sie

MOVE: wenn der Datentransfer in umgekehrter Richtung stattfinden soll.

Siehe auch: MOVEA, PEA

LINK

Reserviere Bereich im Stack
link and allocate

SP - 4 -> SP; An -> (SP)
SP -> An; SP + d16 -> SP

Operandgröße: keine

Assembler Syntax: LINK An, d16
(Adressdifferenz)

Beschreibung:

Die Wert des angegebenen Adressregisters wird auf den Stack gepushed. Danach nimmt das Adressregister den Wert des Stack Pointers an.

Die Adressdifferenz d16, die als eine 16-Bit Binärzahl mit Vorzeichen aufgefaßt wird, wird zu dem Stack Pointer addiert.

Die Adressdifferenz ist normalerweise eine negative Zahl. Sie gibt die Größe des Bereichs an, das im Stack reserviert werden soll.

Der reservierte Bereich innerhalb des Stacks wird als **Stack Frame** bezeichnet, es befindet sich zwischen SP und An. SP stellt die niedrige und An die höhere Grenze dar.

PROGRAMMIERHINWEIS:
Durch die Verwendung von LINK wird das Problem von falschen Rückkehradressen infolge Fehler bei der Stackprogrammierung etwas entschärft. Etwaige Adressierungsfehler innerhalb des von LINK reservierten Bereichs haben nicht mehr ein unbedingtes Abstürzen des Programms zufolge.

Reserviere Bereich im Stack
link and allocate

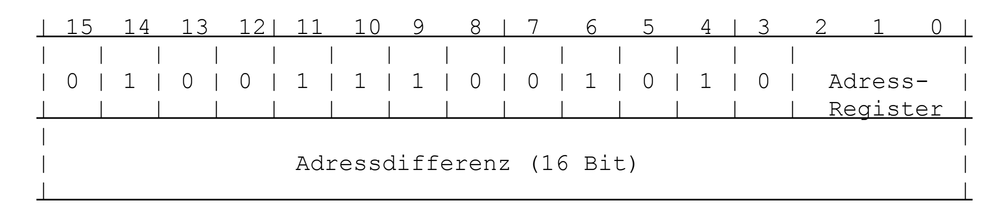
LINK

Der Befehl UNLK macht genau das entgegengesetzte von LINK, damit wird der reservierte Stackbereich wieder freigegeben, auch das Adressregister An erhält seinen früheren Wert wieder.

X	N	Z	V	C
-	-	-	-	-

Condition Code Register: bleibt unverändert

Format des Befehlswortes:



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	0	1	0	Adress-Register		
Adressdifferenz (16 Bit)															

Bit 2..0 Feld Adressregister
spezifiziert, in welchem Adressregister der alte Stack Pointer aufgehoben werden soll.

Feld Adressdifferenz: enthält eine 16-Bit Binärzahl mit Vorzeichen. Dieser Wert wird zu dem Stack Pointer addiert.

LSL

Logisches Schieben nach links
logical shift left

Ziel verschoben durch <Zahl> -> Ziel

Operandgröße: LSL.B Byte (8 Bit)
LSL.W Wort (16 Bit)
LSL.L Langwort (32 Bit)
LSL Wort (16 Bit)

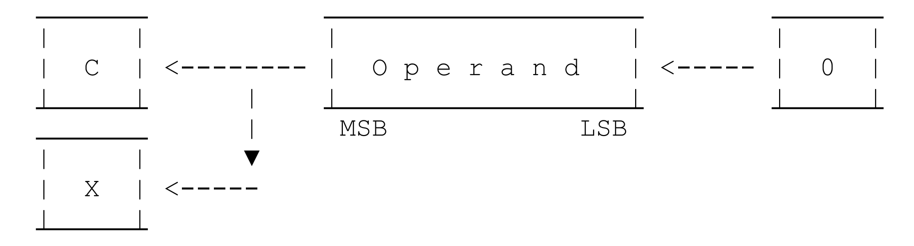
Assembler Syntax:
LSL.x Dn, Dn (Quelle, Ziel)
LSL.x #<data>, Dn
(x entspricht B, W, L)
LSL <ea>

Beschreibung:

Die Bits des Operanden werden nach links verschoben. Bei jedem Schiebeschritt passiert folgendes:

- o Das C-Bit und das X-Bit erhalten den Wert des hochwertigen Bits.
- o Danach erhält das hochwertige Bit den Wert des Bits rechts daneben. Dieses Bit erhält dann den Wert seines rechten Nachbarns usw, bis Bit 1 den Wert von Bit 0 erhält.
- o Zum Schluß erhält Bit 0 den Wert Null.

Für die Gesamtzahl der Schiebeschritte siehe weiter unten.



Logisches Schieben nach links
logical shift left

LSL

Ein LSL um n Positionen bedeutet, daß der Operand - als Binärzahl ohne Vorzeichen aufgefaßt - mit 2^n multipliziert wird.

HINWEIS:
Der Befehl ist weitgehend ähnlich zum Befehl ASL. Der Unterschied ist der, daß das ASL das V-Bit setzt, sobald das hochwertige Bit des Operanden sich während des Verschieben mindestens einmal ändert.

Es gibt drei Befehlsformen.

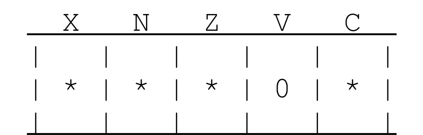
- Der Befehl
  LSL.x t<data>, Dn (x entspricht B, W, L)
  schiebt ein Datenregister nach links um soviele Positionen, wie in der Konstante angegeben ist. Die Maximalzahl ist acht.

- Der Befehl
  LSL.x Dn, Dn (x entspricht B, W, L)
  schiebt ein Datenregister nach links. Ein zweites Datenregister legt fest, um wieviele Positionen geschoben wird.

- Der Befehl
  LSL <ea>
  schiebt eine Speicherstelle (16 Bit) um eine Position nach links.

LSL

Logisches Schieben nach links
logical shift left



X	N	Z	V	C
*	*	*	0	*

Condition Code Register:

C erhält den Wert, der zuletzt aus dem hochwertigen Bit des Operanden herausgeschoben wurde. Wird zurückgesetzt bei Schieben um Null Positionen.
V wird zurückgesetzt.
Z wird gesetzt, wenn das Ergebnis gleich Null ist. Wird sonst zurückgesetzt.
N wird gesetzt, wenn das hochwertige Bit des Ergebnisses gesetzt ist. Wird sonst zurückgesetzt.
X erhält den Wert, der zuletzt aus dem hochwertigen Bit des Operanden herausgeschoben wurde. Bleibt unverändert bei Schieben um Null Positionen.

Logisches Schieben nach links
logical shift left

LSL

Assembler Syntax: LSL.x Dn, Dn
LSL.x #<data>, Dn
(x entspricht B, W, L)

Dazu gehört das folgende Format des Befehlswortes:

15	14	13	12	11 10 9	8	7 6	5	4	3	2 1 0
1	1	1	0	Zähl-Register	1	Größe	i	0	1	Daten-Register

Bit 11..9 Zählregister-Feld
Wenn i = 0:
Die Bits 11..9 geben an, um wieviele Positionen die Bits des Zieloperanden nach links verschoben werden. Dabei entspricht 001 einer Position usw., bis 111 sieben Positionen entspricht. 000 entspricht aber acht Positionen. (Konstante)
Wenn i = l:
Die Bits 11..9 wählen ein Datenregister Dn an. Die niederwertigen 6 Bits des Datenregisters Dn geben an, um wieviele Positionen die Bits des Zieloperanden nach links verschoben werden.

Bit 7..6 Größe-Feld
00 Byte-Befehl LSL.B
01 Wort-Befehl LSL.W
10 Langwort-Befehl LSL.W

Bit 5 i-Feld
0 Die Bits 11..9 beziehen sich auf eine Konstante
l Die Bits 11..9 beziehen sich auf ein Datenregister.

Bit 2..0 Register-Feld
wählt ein Datenregister als Zieloperand an.

LSL

Logisches Schieben nach links
logical shift left

Assembler Syntax: LSL <ea>

Die 16 Bits des angewählten Speicherwortes werden um eine Position nach links verschoben.

Dazu gehört das folgende Format des Befehlswortes:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	0	1	1	1	1	Effektive Adresse					
										Mode			Register		

Bit 5..0 Wählt die Effektive Adresse des Quell-operanden an. Die folgenden Adressierungsarten sind erlaubt:

Adr.-Art	Mode	Reg
Dn	nicht erlaubt	
An	nicht erlaubt	
(An)	010	R:An
(An)+	011	R:An
-(An)	100	R:An
d16(An)	101	R:An
d8(An,Xi)	110	R:An

Adr.-Art	Mode	Reg
xxx.W	111	000
xxx.L	111	001
d16(PC)	nicht erlaubt	
d8(PC,Xi)	nicht erlaubt	
#<data>	nicht erlaubt	
Erläuterung siehe Kapitel 5		

Für die Adressierungsart Dn siehe auf der vorherigen Seite.

Benutzen Sie
LSR: wenn nach rechts geschoben werden soll
ASL: Wenn das V-Bit gesetzt werden soll, sobald das hochwertige Bit des Operanden sich während des Verschiebens mindestens einmal ändert.

Siehe auch: ASR, ROL, ROR, ROXL, ROXR, SWAP

Logisches Schieben nach rechts
logical shift right

LSR

Ziel verschoben durch <Zahl> -> Ziel

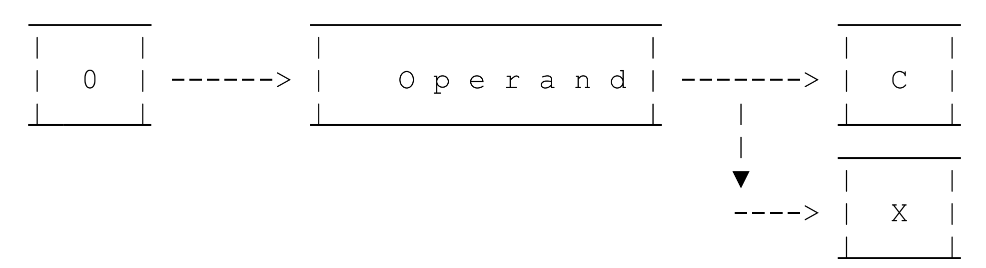
Operandgröße: LSR.B Byte (8 Bit)
LSR.W Wort (16 Bit)
LSR.L Langwort (32 Bit)
LSR Wort (16 Bit)

Assembler Syntax:
LSR.x Dn, Dn (Quelle, Ziel)
LSR.x f<data>, Dn
(x entspricht B, W, L)
LSR <ea>

Beschreibung:
Die Bits des Operanden werden nach rechts verschoben. Bei jedem Schiebeschritt passiert folgendes:

- o Bit 0 gibt seinen Wert an das C-Bit und das X-Bit ab.
- o Danach gibt Bit 1 seinen Wert an Bit 0, und Bit 2 seinen Wert an Bit 1 usw., bis das hochwertige Bit seinen Wert an seinen rechten Nachbarn abgibt.
- o Zum Schluß erhält das hochwertige Bit den Wert Null.

Für die Gesamtzahl der Schiebeschritte siehe weiter unten.



LSR

Logisches Schieben nach rechts
logical shift right

Ein LSR um n Positionen bedeutet, daß der Operand - als Binärzahl ohne Vorzeichen aufgefaßt - durch 2n dividiert wird: der Operand erhält den Wert des Quotienten. Der Wert des Restes ist so nicht feststellbar.

HINWEIS:
Der Befehl ist weitgehend ähnlich zum Befehl LSL.
Der Unterschied ist der, daß das LSL das V-Bit zurücksetzt.

Es gibt drei Befehlsformen.

- Der Befehl
  LSR.x #<data>, Dn (x entspricht B, W, L)
  schiebt ein Datenregister nach rechts um soviele Positionen, wie in der Konstante angegeben ist. Die Maximalzahl ist acht.

- Der Befehl
  LSR.x Dn, Dn (x entspricht B, W, L)
  schiebt ein Datenregister nach rechts. Ein zweites Datenregister legt fest, um wieviele Positionen geschoben wird.

- Der Befehl
  LSR <ea>
  schiebt eine Speicherstelle (16 Bit) um eine Position nach rechts.

Logisches Schieben nach rechts
logical shift right

LSR

Condition Code Register:

X	N	Z	V	C
*	*	*	0	*

C erhält den Wert, der zuletzt aus dem niederwertigen Bit des Operanden herausgeschoben wurde. Wird zurückgesetzt bei Schieben um Null Positionen.
V wird zurückgesetzt.
Z wird gesetzt, wenn das Ergebnis gleich Null ist. Wird sonst zurückgesetzt.
N wird gesetzt, wenn das hochwertige Bit des Ergebnisses gesetzt ist (zeigt ein negatives Ergebnis).
Wird sonst zurückgesetzt.
X erhält den Wert, der zuletzt aus dem niederwertigen Bit des Operanden herausgeschoben wurde. Bleibt unverändert bei Schieben um Null Positionen.

LSR

Logisches Schieben nach rechts
logical shift right

Assembler Syntax: LSR.x Dn, Dn
LSR.x #<data>, Dn
(x entspricht B, W, L)

Dazu gehört das folgende Format des Befehlswortes:

15	14	13	12	11 10 9	8	7 6	5	4	3	2 1 0
1	1	1	0	Zähl-Register	0	Größe	i	0	1	Daten-Register

Bit 11..9 Zählregister-Feld
Wenn i = 0:
Die Bits 11..9 geben an, um wieviele Positionen die Bits des Zieloperanden nach rechts verschoben werden.Dabei entspricht 001 einer Position usw., bis 111 sieben Positionen entspricht. 000 entspricht aber acht Positionen. (Konstante)
Wenn i = l :
Die Bits 11..9 wählen ein Datenregister Dn an. Die niederwertigen 6 Bits des Datenregisters Dn geben an, um wieviele Positionen die Bits des Zieloperanden nach rechts verschoben werden.

Bit 7..6 Größe-Feld
00 Byte-Befehl LSR.B
01 Wort-Befehl LSR.W
10 Langwort-Befehl LSR.W

Bit 5 i-Feld
0 Die Bits 11.. 9 beziehen sich auf eine Konstante
l Die Bits 11..9 beziehen sich auf ein Datenregister.

Bit 2..0 Register-Feld
wählt ein Datenregister als Zieloperand an.

Logisches Schieben nach rechts
logical shift right

LSR

Assembler Syntax: LSR <ea>

Die 16 Bits des angewählten Speicherwortes werden um eine Position nach rechts verschoben.

Dazu gehört das folgende Format des Befehlswortes:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	0	0	0	1	1	Effektive Adresse					
										Mode			Register		

Bit 5..0 Wählt die Effektive Adresse des Quelloperanden an. Die folgenden Adressierungsarten sind erlaubt:

Adr.-Art	Mode	Reg
Dn	nicht erlaubt	
An	nicht erlaubt	
(An)	010	R:An
(An)+	011	R:An
-(An)	100	R:An
d16(An)	101	R:An
d8(An,Xi)	110	R:An

Adr.-Art	Mode	Reg
xxx.W	111	000
xxx.L	111	001
d16(PC)	nicht erlaubt	
d8(PC,Xi)	nicht erlaubt	
#<data>	nicht erlaubt	
Erläuterung siehe Kapitel 5		

Für die Adressierungsart Dn siehe auf der vorherigen Seite.

Benutzen Sie
LSL: wenn nach links geschoben werden soll
ASR: Wenn das hochwertige Bit seinen Wert behalten soll.

Siehe auch: ASL, ROL, ROR, ROXL, ROXR, SWAP

MOVE

Kopiere Daten
move date

Quelle -> Ziel

Operandgröße: MOVE.B Byte (8 Bit)
MOVE.W Wort (16 Bit)
MOVE.L Langwort (32 Bit)

Assembler Syntax: MOVE.x <ea>, <ea> (Quelle, Ziel)
(x entspricht B, W, L)

Beschreibung:

Der Quelloperand wird in den Zieloperand kopiert. Die Daten werden geprüft, und das Condition Code Register wird entsprechend gesetzt. Die Operandgröße kann Byte, Wort oder Langwort sein.

Condition Code Register:

X	N	Z	V	C
-	*	*	0	0

C wird zurückgesetzt.
V wird zurückgesetzt.
Z wird gesetzt, wenn das Ergebnis gleich Null ist.
Wird sonst zurückgesetzt.
N wird gesetzt, wenn das Ergebnis negativ ist.
Wird sonst zurückgesetzt.
X bleibt unverändert

Kopiere Daten
move date

MOVE

Format des Befehlswortes:

15	14	13 12	11 10 9	8 7 6	5 4 3	2 1 0
1	1	Größe	Z i e l		Q u e l l e	
			Register	Mode	Mode	Register

Bit 13..12 Größe-Feld:
 01 Byte-Befehl MOVE.B
 11 Wort-Befehl MOVE.W
 10 Langwort-Befehl MOVE.L

Bit 11..6 Effektive Adresse des Zieloperanden, folgenden Adressierungsarten sind erlaubt:

Adr.-Art	Mode	Reg
Dn	000	R:Dn
An	nicht erlaubt	
(An)	010	R:An
(An)+	011	R:An
-(An)	100	R:An
d16(An)	101	R:An
d8(An,Xi)	110	R:An

Adr.-Art	Mode	Reg
xxx.W	111	000
xxx.L	111	001
d16(PC)	nicht erlaubt	
d8(PC,Xi)	nicht erlaubt	
#<data>	nicht erlaubt	
Erläuterung siehe Kapitel 5		

MOVE

Kopiere Daten
move date

Bit 5..0 Effektive Adresse des Quelloperanden, folgenden Adressierungsarten sind erlaubt:

Adr.-Art	Mode	Reg
Dn	000	R:Dn
An *)	001	R:An
(An)	010	R:An
(An)+	011	R:An
-(An)	100	R:An
d16(An)	101	R:An
d8(An,Xi)	110	R:An

Adr.-Art	Mode	Reg
xxx.W	111	000
xxx.L	111	001
d16(PC)	111	010
d8(PC,Xi)	111	011
#<data>	111	100
Erläuterung siehe Kapitel 5		

*) Adressierungsart An nicht für Byte-Befehle erlaubt

Benutzen Sie

MOVEA:	wenn der Zieloperand ein Adressregister ist.
MOVEI oder MOVEQ:	wenn der Quelloperand eine Konstante ist.
MOVEP:	wenn Sie nur auf geraden oder ungeraden Adressen zugreifen.
EXG:	Wenn Sie Daten vertauschen möchten.
MOVE zum CCR	wenn Sie das CCR ändern möchten.
MOVE vom SR	wenn Sie das SR lesen möchten.
MOVE zum SR	wenn Sie das SR ändern möchten.
MOVE USP	wenn Sie das USP lesen oder ändern möchten.

Kopiere zum Condition Code Register
move to the Condition Code Register

MOVE zum CCR

Quelle -> CCR

Operandgröße: Wort (16 Bit)

Assembler Syntax: MOVE <ea>, CCR (Quelle, Ziel)

Beschreibung:
Das niederwertige Byte des Quelloperanden wird in das Condition Code Register kopiert. Die Operandgröße ist ein Wort. Das hochwertige Byte wird nicht benutzt.

Condition Code Register:

X	N	Z	V	C
*	*	*	*	*

C erhält den Wert von Bit 0 des Quelloperanden
V erhält den Wert von Bit 1 des Quelloperanden
Z erhält den Wert von Bit 2 des Quelloperanden
N erhält den Wert von Bit 3 des Quelloperanden
X erhält den Wert von Bit 4 des Quelloperanden

ACHTUNG:
MOVE zum CCR ist ein Wort-Befehl.
Die Befehle ANDI zum CCR,
ORI zum CCR und
EORI zum CCR
dagegen sind Byte-Befehle.

MOVE zum CCR

Kopiere zum Condition Code Register
move to the Condition Code Register

Format des Befehlswortes:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	1	1	Effektive Adresse					
										Mode			Register		

Bit 5..0 Effektive Adresse des Quelloperanden,
folgenden Adressierungsarten sind erlaubt:

Adr.-Art	Mode	Reg
Dn	000	R:Dn
An	nicht erlaubt	
(An)	010	R:An
(An)+	011	R:An
-(An)	100	R:An
d16(An)	101	R:An
d8(An,Xi)	110	R:An

Adr.-Art	Mode	Reg
xxx.W	111	000
xxx.L	111	001
d16(PC)	111	010
d8(PC,Xi)	111	011
#<data>	111	100
Erläuterung siehe Kapitel 5		

Benutzen Sie

HOVE vom SR: wenn Sie auch das System Byte einlesen möchten.

MOVE zum SR: Wenn Sie das CCR und/oder das System Byte ändern möchten

ANDI zum CCR:
ORI zum CCR:
EORI zum CCR: wenn Sie nur bestimmte Bits des CCR ändern möchten.

Siehe auch: MOVE, MOVEA, MOVE USP

Kopiere zum Status Register
move to the Status Register
(privilegierter Befehl)

MOVE zum SR

Wenn Supervisor Mode: Quelle -> SR

Wenn User Mode: **Auslösung Exception 8 (Kap 6)**
(Verletzung Privilegium)

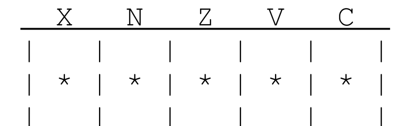
Operandgröße: Wort (16 Bit)

Assembler Syntax: MOVE <ea>, SR (Quelle, Ziel)

Beschreibung:
Wenn sich der Prozessor im Supervisor Mode befindet, wird der Quelloperand in das Status Register kopiert. Die Operandgröße ist ein Wort.

Wenn der Prozessor sich dagegen in User Mode befindet, findet ein Trap statt.

Das Status Register wird im Kap. 2 beschrieben.



X	N	Z	V	C
*	*	*	*	*

Condition Code Register:

c erhält den Wert von Bit 0 des Quell Operanden
v erhält den Wert von Bit 1 des Quell Operanden
z erhält den Wert von Bit 2 des Quell Operanden
N erhält den Wert von Bit 3 des Quell Operanden
X erhält den Wert von Bit 4 des Quell Operanden

MOVE zum SR

Kopiere zum Status Register
move to the Status Register

Format des Befehlswortes:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	1	0	1	1	Effektive Adresse					
										Mode			Register		

Bit 5..0 Effektive Adresse des Quelloperanden. Die folgenden Adressierungsarten sind erlaubt:

Adr.-Art	Mode	Reg
Dn	000	R:Dn
An	nicht erlaubt	
(An)	010	R:An
(An)+	011	R:An
-(An)	100	R:An
d16(An)	101	R:An
d8(An,Xi)	110	R:An

Adr.-Art	Mode	Reg
xxx.W	111	000
xxx.L	111	001
d16(PC)	111	010
d8(PC,Xi)	111	011
#<data>	111	100
Erläuterung siehe Kapitel 5		

Benutzen Sie

MOVE vom SR: wenn Sie das CCR und/oder das System Byte einlesen möchten. (Priviligierter Befehl)

MOVE zum CCR: wenn Sie nur das CCR ändern möchten.

Siehe auch: MOVE, MOVEA, MOVE USP

Kopiere vom Status Register
move from the Status Register

MOVE vom SR

SR -> Ziel

Operandgröße: Wort (16 Bit)

Assembler Syntax: MOVE SR, <ea> (Quelle, Ziel)

Beschreibung:
Der Inhalt des Status Registers wird in den Ziel-Operand kopiert. Die Operandgröße ist ein Wort.

Das Status Register wird im Kap. 2 beschrieben.

Hinweis: auf den Prozessoren 68010 und 68012 verhält sich dieser Befehl abweichend. MOVE vom SR ist dort ein privilegierter Befehl, so daß ein Trap erfolgt, wenn der Prozessor nicht in Supervisor Mode ist. Der dort verfügbare nicht-privilegierte Befehl MOVE vom CCR ist auf "unserem" Prozessor 68000 nicht vorhanden.

X	N	Z	V	C
-	-	-	-	-

Condition Code Register:
keine Änderungen

MOVE vom SR

Kopiere vom Status Register
move from the Status Register

Format des Befehlswortes:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	0	1	1	Effektive Adresse					
										Mode			Register		

Bit 5..0 Effektive Adresse des Zieloperanden, folgenden Adressierungsarten sind erlaubt:

Adr.-Art	Mode	Reg
Dn	000	R:Dn
An	nicht erlaubt	
(An)	010	R:An
(An)+	011	R:An
-(An)	100	R:An
d16(An)	101	R:An
d8(An,Xi)	110	R:An

Adr.-Art	Mode	Reg
xxx.W	111	000
xxx.L	111	001
d16(PC)	nicht erlaubt	
d8(PC,Xi)	nicht erlaubt	
#<data>	nicht erlaubt	
Erläuterung siehe Kapitel 5		

Benutzen Sie

MOVE zum SR: wenn Sie das CCR und/oder das Sytem Byte ändern möchten. (Privilegierter Befehl)

MOVE zum CCR: wenn Sie nur das CCR ändern möchten.

Siehe auch: MOVE, MOVEA, MOVE USP, ANDI zum SR, ORI zum SR

Kopiere vom/zum User Stack Pointer
move User Stack Pointer
(privilegierter Befehl)

MOVE USP

Wenn Supervisor Mode: USP -> An bzw.
An -> ÜSP

Wenn User Mode: Auslösung Exception 8 (Kap 6)
(Verletzung Privilegium)

Operandgröße: Langwort (32 Bit)

Assembler Syntax:
MOVE USP, An (Quelle, Ziel)
MOVE An, USP

Beschreibung:
Wenn sich der Prozessor im Supervisor Mode befindet, wird der Inhalt des User Stack Pointers zum oder vom Adressregister kopiert.

Wenn der Prozessor sich dagegen in User Mode befindet, findet ein Trap statt.

Der Supervisor Mode wird im Kap. 2 beschrieben.

X	N	Z	V	C
-	-	-	-	-

Condition Code Register:
keine Änderungen

MOVE USP

Kopiere vom/zum User Stack Pointer
move User Stack Pointer
(privilegierter Befehl)

Format des Befehlswortes:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	0	d	Adress-Register		

Bit 3   Feld d
spezifiziert die Richtung der Operation
 0 kopieren vom Adressregister zum User Stack Pointer
 1 kopieren vom User Stack Pointer zum Adressregister

Bit 2..0 Registerfeld
wählt das Adressregister an.

Benutzen Sie

MOVEA: Wenn Sie auf ein anderes Datenregister Zugrigff nehmen möchten.

MOVE vom SR: wenn Sie auch das System Byte einlesen möchten.

MOVE zum SR: Wenn Sie das CCR und/oder das System Byte ändern möchten. (Privilegierter Befehl)

MOVE zum CCR: Wenn Sie nur das System Byte ändern möchten.

Siehe auch: MOVE, MOVEA, MOVE USP

Kopiere Daten in ein Adress-Register
move address

MOVEA

Quelle -> Ziel

Operandgröße: MOVEA.W Wort (16 Bit)
MOVEA.L Langwort (32 Bit)

Assembler Syntax: MOVEA.x <ea>, An (Quelle, Ziel)
(x entspricht W, L)

Beschreibung:

Der Quelloperand wird in das Zieloperand-Adressregister kopiert. Die Operandgröße kann Wort oder Langwort sein.
Wenn der Operand ein Wort ist, wird das Vorzeichen auf 32 Bit erweitert, bevor die Operation stattfindet.

X	N	Z	V	C
-	-	-	-	-

Condition Code Register:
keine Änderungen

MOVEA

Kopiere Daten in ein Adress-Register
move address

Format des Befehlswortes:

```
| 15  14  13  12| 11  10  9   8 | 7   6   5   4 | 3   2   1   0 |
|   |   |       |           |   |   |   |                       |
| 0 | 0 | Größe |  Register | 0 | 0 | 1 |    Effektive Adresse  |
|   |   |       |     An    |   |   |   |    Mode    |  Register|
```

Bit 13..12 Größe-Feld:
 11 Wort-Befehl MOVEA.W Das Vorzeichen des Operanden wird auf 32 Bit erweitert, bevor die Operation stattfindet.
 10 Langwort-Befehl MOVEA.L

Bit 11..9 Wählt den Zieloperand (Adressregister) an.

Bit 5..0 Effektive Adresse des Quelloperanden. Die folgenden Adressierungsarten sind erlaubt:

Adr.-Art	Mode	Reg
Dn	000	R:Dn
An	001	R:An
(An)	010	R:An
(An)+	011	R:An
-(An)	100	R:An
d16(An)	101	R:An
d8(An,Xi)	110	R:An

Adr.-Art	Mode	Reg
xxx.W	111	000
xxx.L	111	001
d16(PC)	111	010
d8(PC,Xi)	111	011
#<data>	111	100
Erläuterung siehe Kapitel 5		

Benutzen Sie
MOVE: wenn der Zieloperand kein Adressregister ist;
MOVEI oder MOVEQ: wenn der Quelloperand eine Konstante ist.

Siehe auch: MOVEM, MOVEP, LEA

Kopiere mehrere Register
move multiple registers

MOVEM

Register -> Ziel bzw. Quelle -> Register

Operandgröße: MOVEM.W Wort (16 Bit)
MOVEM.L Langwort (32 Bit)

(Quelle, Ziel)

Assembler Syntax:
MOVEM.x Registerliste, <ea>
MOVEM.x <ea>, Registerliste
(x entspricht B, W, L)

Beschreibung:

Es gibt zwei Übertragungsrichtungen:
- o entweder werden die angewählten Register zum Speicher kopiert;
- o oder die im Speicher abgelegten Registerkopien werden in die Register kopiert.

Der im Speicher belegte Platz für die Registerkopien ist zusammenhängend. Die Anfangsadresse im Speicher wird von der effektiven Adresse im Befehl angegeben.
Im Befehl wird auch angegeben, ein wie großes Stück von jedem Register kopiert wird: entweder die gesamten Register als Langwort (MOVEM.L), oder nur die niederwertigen Hälften der Register als 16-Bit Wort (MOVEM.W). Wenn jeweils ein 16-bit Wort aus dem Speicher zu einem Register übertragen wird, wird das Vorzeichen auf 32 Bit erweitert (auch bei Datenregistern) und im Register übertragen.

Mit dem Befehl MOVEM kann man schnell Registerinhalte retten und wieder instandsetzen. Bei Interrupt- oder Exception Behandlungen, wo es manchmal nicht bekannt ist, welche Register geändert werden, sind normalerweise alle Register zu retten.

MOVEM

Kopiere mehrere Register
move multiple registers

ACHTUNG: EXTRA SPEICHERZUGRIFF
MOVEM macht einen zusätzlichen, für den Programmierer unerwarteten Lesevorgang im Speicher, und zwar auf der Adresse direkt nach der letzten Registerkopie.

Normalerweise ist das unwichtig, aber wenn Sie Ihre Register gerade auf der allerletzten verfügbaren Speicherstelle ablegen möchten, verursacht der Zugriff auf einer nicht bestehenden Speicheradresse einen Busfehler, der einen Trap bewirkt. Wenn Sie den Fehler nicht abfangen, wird Ihr Programm abstürzen.

BEISPIEL 1:
Der Befehl

```
MOVEM.W  D2/D5, $1000
```

speichert die niederwertigen Hälften der Register D2 und D5 auf den Adressen $1000 und $1002 ab. (Bei einer Argumentlänge von 2 Bytes liegt die nächste Adresse 2 Bytes weiter.)

Der Befehl

```
MOVEM.W  $1000, D2/D5
```

ladet die niederwertigen Hälften der Register D2 und D5 mit den Inhalten der Speicheradressen $1000 und $1002. Er bewirkt für die niederwertigen Registerhälften von D2 und D5 genau das entgegengesetzte von dem ersten Befehl.

Es findet ein extra Speicherzugriff auf der Adresse $1004 statt.

Kopiere mehrere Register
move multiple registers

MOVEM

BEISPIEL 2:

Der Befehl

```
MOVEM.L  D0-D7/A0-A7, $2000
```

legt Kopien von allen Registern im Speicher ab. Der Speicherbereich ist zusammenhängend. Das erste Register wird auf der Adresse $2000 abgelegt. Weil MOVEM.L eine Argumentlänge von 4 Bytes hat, wird das zweite Register, das in der Liste enthalten ist, auf der Adresse $2004 abgelegt, usw.

Insgesamt werden die Register so abgelegt:

```
$2000  D0     $2010  D4      $2020  A0     $2030  A4
$2004  Dl     $2014  D5      $2024  AI     $2034  A5
$2008  D2     $2018  D6      $2028  A2     $2038  A6
$200C  D3     $201C  D7      $202C  A3     $203C  A7
```

Achtung! Extra Lesezugriff auf der Adresse $2040

Der Befehl

```
MOVEM.L  $2000, D0-D7/A0-A7
```

bringt die ursprünglichen Registerinhalte wieder in die Register zurück.

BEISPIEL 3:

Sie können Register auf den Stack ablegen mit

```
MOVEM.L D0-D7/A0-A6, -(SP)
```

und hervorrufen mit

```
MOVEM.L (SP)+, D0-D7/A0-A6
```

MOVEM

Kopiere mehrere Register
move multiple registers

DATENFORMAT IM SPEICHER

Wie die Daten im Speicher abgelegt oder zurückgelesen werden, hängt vom Adressierungsmode ab.

- o Wenn MOVEM in der Postinkrementmode (An)+ adressiert, ist nur ein Datentransfer vom Speicher zu den Registern erlaubt. Die Register werden ab der angegebenen Speicherstelle geladen bis zur höchsten Adresse. Die Reihenfolge der Daten ist von Datenregister D0 bis D7, und dann von Adressregister A0 bis A7. Nach Beendigung des Befehls enthält das Adressregister die Adresse des letzten Registers plus die Operandlänge in Byte (2 oder 4).

- o Wenn MOVEM in der Prädekrementmode -(An) adressiert, ist nur ein Datentransfer von den Registern zum Speicher erlaubt. Das erste Register wird abgespeichert an der spezifizierten Adresse minus der Operandlänge in Bytes (2 oder 4) bis zur niedrigsten Adresse. Die Reihenfolge vom Abspeichern ist von Adressregister A7 bis A0, und dann von Datenregister D7 bis D0. Nach Beendigung des Befehls enthält das Adressregister die Adresse des zuletzt gespeicherten Registers.

- o Wenn MOVEM in einer der anderen Moden adressiert (siehe Liste), werden die Register gespeichert oder geladen ab der angegeben Adresse bis zur höchsten Adresse. Die Reihenfolge der Daten ist von Datenregister D0 bis D7, und dann von Adressregister A0 bis A7.

Kopiere mehrere Register
move multiple registers

MOVEM

X	N	Z	V	C
-	-	-	-	-

Condition Code Register:
keine Änderungen

Format des Befehlswortes:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	d	0	0	1	g	Effektive Adresse Mode			Register		
Maske mit Registerliste															

Bit 10 d-Feld
bestimmt die Richtung der Datenübertragung
 0 von Register zu Speicher
 1 von Speicher zu den Registern

Bit 6 g-Feld
bestimmt die Größe der zu übertragenen Daten
 0 MOVEM.W Wort-Befegl
 1 MOVEM.L Langwort-Befehl

MOVEM

Kopiere mehrere Register
move multiple registers

Bit 5..0 Effektive Adresse
spezifiziert die Speicheradresse, an welcher Stelle die Datenübertragung anfangen soll.

Für Datenübertragungen von den Registern zum Speicher sind die folgenden Adressierungsarten erlaubt:

Adr.-Art	Mode	Reg
Dn	nicht erlaubt	
An	nicht erlaubt	
(An)	010	R:An
(An)+	nicht erlaubt	
-(An)	100	R:An
d16(An)	101	R:An
d8(An,Xi)	110	R:An

Adr.-Art	Mode	Reg
xxx.W	111	000
xxx.L	111	001
d16(PC)	nicht erlaubt	
d8(PC,Xi)	nicht erlaubt	
#<data>	nicht erlaubt	
Erläuterung siehe Kapitel 5		

Für Datenübertragungen vom Speicher zu den Registern sind die folgenden Adressierungsarten erlaubt:

Adr.-Art	Mode	Reg
Dn	nicht erlaubt	
An	nicht erlaubt	
(An)	010	R:An
(An)+	011	R:An
-(An)	nicht erlaubt	
d16(An)	101	R:An
d8(An,Xi)	110	R:An

Adr.-Art	Mode	Reg
xxx.W	111	000
xxx.L	111	001
d16(PC)	111	010
d8(PC,Xi)	111	011
#<data>	nicht erlaubt	
Erläuterung siehe Kapitel 5		

Kopiere mehrere Register
move multiple registers

MOVEM

Argumentwort: Maske mit Registerliste spezifiziert, welche Register übertragen werden sollen. Ein- oder ausgeschlossen werden kann jeder der Register A0..A7 und D0..D7.

Ein Register wird übertragen, wenn das entsprechende Bit gesetzt ist. Es wird nicht übertragen, wenn das Bit zurückgesetzt ist.

Für den Prädekrement-Mode .-(An) ist die Maske so aufgebaut:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
D0	D1	D2	D3	D4	D5	D6	D7	A0	A1	A2	A3	A4	A5	A6	A7

Für alle anderen Adressierungsmethoden ist die Maske so aufgebaut:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
A7	A6	A5	A4	A3	A2	A1	A0	D7	D6	D5	D4	D3	D2	D1	D0

Das Register, das dem niederwertigen Bit entspricht, wird zuerst übertragen, und das Register, das dem hochwertigen Bit entspricht, zuletzt.

MOVEP

Eingabe/Ausgabe zur/von Peripherie
move periperal data

Quelle -> Ziel

Operandgröße: MOVEP.W Wort (16 Bit)
MOVEP.L Langwort (32 Bit)

Assembler Syntax: (Quelle, Ziel)

```
MOVEP.x Dn, d16(An)  oder  MOVEP.x Dn, (d16,An)
MOVEP.x d16(An),Dn   oder  MOVEP.x (d16,An),Dn
                                (x entspricht B,W,L)
```

Beschreibung:

Der Befehl MOVEP ist eine bequeme Methode, um auf 8-Bit Eingabe/Ausgabe-Kanäle (Input/Output) zuzugreifen.

Der Prozessor 68000 ist ein 16-Bit Prozessor. Das bedeutet konkret, daß der Microprozessor-IC 68000, der sich in Ihrem Computer befindet, unter anderem 16 Daten-Anschlüsse D0..D15 besitzt. Auf Ihrer Platine werden diese Daten-Anschlüsse dann vom IC-Sockel über Leiterbahnen mit den dort vorhandenen Speicher-ICs verbunden.

> Ob Ihre Speicherbausteine eine Datenbreite von 1, 4, 8, oder 16 Bit haben, ist dabei unerheblich. Es werden soviele Speicherbausteine genommen, daß alle 16 Datenleitungen des Prozessors an irgendeinen Speicher-IC angeschlossen sind.

Eingabe/Ausgabe zur/von Peripherie
move periperal data

MOVEP

Auf dem Bild auf der nächsten Seite ist eine Darstellung für Speicherbausteine mit 8-Bit Datenbreite gegeben.

Die ICs für Ein- und Ausgabe sind an die gleichen Datenleitungen wie die Speicher-ICs angeschlossen.

Viele ICs für Ein- und Ausgabe haben aber eine Datenbus-Breite von 8 Bit. Das ist teils aus historischen Gründen, teils auch, um Anschlüsse zu sparen. Die ICs sind dadurch kleiner und Ihr Computer auch.

Ein IC mit 8 Datenanschlüssen D0..D7 kann ich auf zwei Arten anschließen:

- Ich kann die Datenanschlüsse D0..D7 des IC mit den Datenleitungen D0..D7 des Prozessors verbinden. Das IC befindet sich dann an den ungeraden Speicheradressen 1,3, 5 etc.

- Ich kann aber auch die Datenanschlüsse D0..D7 mit den Datenleitungen D8..D15 des Prozessors verbinden. Das IC befindet sich dann an den geraden Speicheradressen 0, 2, 4 etc.

Die Situation der ICs entspricht den beiden Möglichkeiten in dem Bild auf der nächsten Seite.

Und der Befehl MOVEP macht es uns bequem, um nur auf die geraden oder auf die ungeraden Adressen zuzugreifen.

MOVEP

Eingabe/Ausgabe zur/von Peripherie
move periperal data

```
 _______
|      |                    S  P  E  I  C  H  E  R
|  P   |                       ___________________
|  r   | /|_______________|\ |                   | _______
|  o   |/  Datenleitungen  \|  Oberes Byte    |/       |
|  z   |\ ____D8..D15_____ /|     (8-Bit)      |\__    |
|  e   | \|               |/ |_gerade Adressen_|  |    |
|  s   |                                          |    |
|  s   |                                          |    |
|  o   |                       ___________________ |    |
|  r   | /|_______________|\ |                   | _|    |
|  -   |/  Datenleitungen  \|  Unteres Byte   |/       |
|  I   |\ ____D0..D7______ /|      (8-Bit)     |\__    |
|  C   | \|               |/ |ungerade Adressen|  |    |
|      |                                          |    |
|      |                                          |    |
|      |                                          |    |
|      |                                          |    |
|  6   | /|_______________________________________|    |
|  8   |/  Adressleitungen                             |
|  0   |\ ____A1..A23__________________________________|
|  0   | \|
|  0   |
|      |
|______|
```

Bild B1   Gerade und ungerade Speicherstellen

Eingabe/Ausgabe zur/von Peripherie
move periperal data

MOVEP

MOVEP überträgt Daten zwischen einem Datenregister und jeder geraden oder ungeraden Adresse im Speicher. Die Übertragung fängt an der spezifizierten Speicheradresse an, die Adresse wird nach der Übertragung jeweils um zwei vergrößert. Das hochwertige Byte des Datenregisters wird zuerst übertragen, das niederwertige Byte zuletzt.

Die Speicheradresse wird angewählt mit dem Adressregister indirekte Adressierung mit Adressdifferenz dl6(An).

Wenn die Adresse gerade ist, finden alle Übertragungen zu den Datenleitungen D8..D15 statt, wenn die Adresse ungerade ist, zu den Datenleitungen D0..D7.

Die Anwendung von MOVEP ist nicht auf Prozessoren mit einer Datenbus-Breite von 16 Bit beschränkt. Auch bei Prozessoren mit einer Busbreite von 8 oder 32 Bit bewirkt MOVEP, daß jedes zweite Byte angesprochen wird.

Für die Diskussion der hochwertigen und niederwertigen Bits und Bytes und die geraden und ungeraden Speicherstellen siehe auch Kap. 3

MOVEP

Eingabe/Ausgabe zur/von Peripherie
move periperal data

BEISPIEL 1: 32-Bit Datenübertragung von einem Datenregister zu einer geraden Speicheradresse

```
MOVEP.L  Dl, $1000
```

Byte-Organisation im Register Dl:

```
31          24 23         16 15          8 7           0
|            |            |            |            |
| hochwertig |mitte-hochw.|mitte-niedw.|niederwertig|
|____________|____________|____________|____________|
```

Byte-Organisation im Speicher:

```
          15               8 7               0
          |                |                 |
1000      |   hochwertig   |                 |
          |________________|_________________|
          |                |                 |
1002      | mitte-hochwertig |               |
          |________________|_________________|
          |                |                 |
1004      |mitte-niederwertig|               |
          |________________|_________________|
          |                |                 |
1006      |  niederwertig  |                 |
          |________________|_________________|
```

Eingabe/Ausgabe zur/von Peripherie
move periperal data

MOVEP

BEISPIEL 2: 16-Bit Datenübertragung von einer ungeraden Speicheradresse zum Datenregister

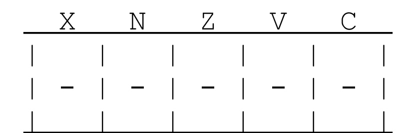
```
MOVEP.W  $2001, D2
```

Byte-Organisation im Speicher:

```
          15               8 7               0
         ______________________________________
         |                 |                 |
2000     |                 |   hochwertig    |
         |_________________|_________________|
         |                 |                 |
2002     |                 |  niederwertig   |
         |_________________|_________________|
```

Byte-Organisation im Register D2:

```
31          24 23        16 15          8 7           0
_______________________________________________________
|            |            |            |            |
|            |            | hochwertig |niederwertig|
|____________|____________|____________|____________|
```



Condition Code Register:
keine Änderung

MOVEP

Eingabe/Ausgabe zur/von Peripherie
move periperal data

Format des Befehlswortes:

15	14	13	12	11 10 9	8 7 6	5	4	3	2 1 0
0	0	0	0	Daten-Register	Operations-Mode	0	0	1	Adress-Register
Adress-differenz									

Bit 11..9 Datenregisterfeld
spezifiziert von oder zu welchem Datenregister die Daten übertragen werden sollen.

Bit 8..6 Feld Operationsmode

```
 100  MOVEP.W  d16(An), Dn
                16-Bit Übertragung Speicher -> Register
 101  MOVEP.L  d16(An), Dn
                32-Bit Übertragung Speicher -> Register
 110  MOVEP.W  Dn, d16(An)
                16-Bit Übertragung Register -> Speicher
 111  MOVEP.L  Dn, d16(An)
                32-Bit Übertragung Register -> Speicher
```

Bit 2..0 Feld Datenregister
spezifiziert, welches Adressregister für die Adressregister indirekte Adressierung mit Adressdifferenz d16(An) benutzt werden soll.

Feld Adress-differenz
spezifiziert die Adressdifferenz beim Ermitteln der Operandadresse.

Benutzen Sie

MOVE:	wenn jede Speicheradresse angespochen werden soll.

kopiere Konstante "quick" (8-Bit)
move quick

MOVEQ

Konstante -> Ziel

Operandgröße: Langwort (32 Bit)

Assembler Syntax: MOVEQ #<data>, Dn

Beschreibung:
Der unmittelbare Wert im Operand (die Konstante) wird zum Ziel-Datenregister übertragen.

Die Konstante muß zwischen -128..127 liegen.

Für die Binärzahl mit Vorzeichen steht im Operand ein Feld von 8 Bit zur Verfügung. Das Vorzeichen wird auf 32 Bit erweitert, bevor die Übertragung zum Datenregister stattfindet.

Condition Code Register:

X	N	Z	V	C
-	*	*	0	0

C wird zurückgesetzt.
V wird zurückgesetzt.
Z wird gesetzt, wenn das Ergebnis gleich Null ist. Wird sonst zurückgesetzt.
N wird gesetzt, wenn das Ergebnis negativ ist. Wird sonst zurückgesetzt.
X bleibt unverändert

MOVEQ

kopiere Konstante "quick" (8-Bit)
move quick

Format des Befehlswortes:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	Daten-Register			0	D a t e n							

Bit 11..9 Registerfeld:
wählt das Datenregister Dn an, zu dem die Konstante übertragen werden soll.

Bit 7..0 Datenfeld:
hier stehen die Daten, die zum Datenregister übertragen werden sollen. Das Vorzeichen wird vor der Übertragung auf 32 Bit erweitert.

Benutzen Sie

MOVE:	wenn die Konstante größer als 8 Bit ist, oder wenn Sie Register oder Speicherinhalte kopieren möchten.

Siehe auch: MOVEM, MOVEP, MOVEQ

Multipliziere mit Vorzeichen
signed multiply



Quelle * Ziel -> Ziel (16 * 16 -> 32)

Operandgröße: Eingangswerte: Wort (16 Bit)
Ergebniswert: Langwort (32 Bit)

Assembler
Syntax: MULS <ea>, Dn (Quelle, Ziel)

Beschreibung:
Zwei Binärzahlen mit Vorzeichen werden multipliziert. Das Ergebnis ist das Produkt der beiden Zahlen unter Berücksichtigung des Vorzeichens.

Der Multiplikator und der Multiplikant sind beide Wörter (16 Bit), das Ergebnis ist ein Langwort (32 Bit).

Vom Register-Operand Dn wird nur die niederwertige Hälfte angewandt, das Ergebnis belegt alle 32 Bits des Register-Operanden Dn.

X	N	Z	V	C
-	*	*	0	0

Condition Code Register:

C wird zurückgesetzt.
V wird zurückgesetzt.
Z wird gesetzt, wenn das Ergebnis gleich Null ist. Wird sonst zurückgesetzt.
N wird gesetzt, wenn das Ergebnis negativ ist. Wird sonst zurückgesetzt.
X bleibt unverändert.

MULS

Multipliziere mit Vorzeichen
signed multiply

Format des Befehlswortes:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	Daten-Register			1	1	1	Effektive Adresse Mode			Effektive Adresse Register		

Bit 11..9 Registerfeld: wählt eines der acht Datenregister als Zieloperand an.

Bit 5..0 Wählt die Effektive Adresse des Quelloperanden an. Die folgenden Adressierungsarten sind erlaubt:

Adr.-Art	Mode	Reg
Dn	000	R:Dn
An	nicht erlaubt	
(An)	010	R:An
(An)+	011	R:An
-(An)	100	R:An
d16(An)	101	R:An
d8(An,Xi)	110	R:An

Adr.-Art	Mode	Reg
xxx.W	111	000
xxx.L	111	001
d16(PC)	111	010
d8(PC,Xi)	111	011
#<data>	111	100
Erläuterung siehe Kapitel 5		

Benutzen Sie
MULU: wenn das Vorzeichen bei der Multiplikation nicht berücksichtigt werden soll.

Siehe auch: DIVS, DIVU

Multipliziere ohne Vorzeichen
unsigned multiply



Quelle * Ziel -> Ziel (16 * 16 -> 32)

Operandgröße: Eingangswerte: Wort (16 Bit)
Ergebniswert: Langwort (32 Bit)

Assembler Syntax: MULU <ea>, Dn (Quelle, Ziel)

Beschreibung:
Zwei Binärzahlen ohne Vorzeichen werden multipliziert. Das Ergebnis ist das Produkt der beiden Zahlen ohne Berücksichtigung des Vorzeichens.

Der Multiplikator und der Multiplikant sind beide Wörter (16 Bit) , das Ergebnis ist ein Langwort (32 Bit).

Vom Register-Operand Dn wird nur die niederwertige Hälfte angewandt, das Ergebnis belegt alle 32 Bits des Register-Operanden Dn.

X	N	Z	V	C
-	*	*	0	0

Condition Code Register:

C wird zurückgesetzt.
V wird zurückgesetzt.
Z wird gesetzt, wenn das Ergebnis gleich Null ist. Wird sonst zurückgesetzt.
N wird gesetzt, wenn das Ergebnis negativ ist. Wird sonst zurückgesetzt.
X bleibt unverändert.



Multipliziere ohne Vorzeichen
unsigned multiply

Format des Befehlswortes:

15	14	13	12	11 10 9	8	7	6	5 4	3 2 1 0
1	1	0	0	Daten-Register	0	1	1	Effektive Adresse Mode	Effektive Adresse Register

Bit 11..9 Registerfeld: wählt eines der acht Datenregister als Zieloperand an.

Bit 5..0 Wählt die Effektive Adresse des Quelloperanden an. Die folgenden Adressierungsarten sind erlaubt:

Adr.-Art	Mode	Reg
Dn	000	R:Dn
An	nicht erlaubt	
(An)	010	R:An
(An)+	011	R:An
-(An)	100	R:An
d16(An)	101	R:An
d8(An,Xi)	110	R:An

Adr.-Art	Mode	Reg
xxx.W	111	000
xxx.L	111	001
d16(PC)	111	010
d8(PC,Xi)	111	011
#<data>	111	100
Erläuterung siehe Kapitel 5		

Benutzen Sie
MULS: wenn das Vorzeichen bei der Multiplikation berücksichtigt werden soll.

Siehe auch: DIVS, DIVU

Negiere BCD-Zahl mit Extend-Bit
negate decimal with extend

NBCD

0 - $Ziel_{10}$ - X -> Ziel

Operandgröße: NBCD.B Byte (8 Bit)
(ein Byte enthält zwei BCD-Zahlen)

Assembler Syntax: NBCD.B <ea>

Beschreibung:
Der Quelloperand und das X-Bit werden von Null subtrahiert: das Ergebnis wird im Zieloperand abgespeichert.

Die Subtrahierung findet als BCD-Arithmetik statt.

Das Ergebnis ist das Zehnerkomplement des Zieloperanden, wenn das X-Bit gesetzt ist, und das Neunerkomplement, wenn das X-Bit zurückgesetzt ist.

Für eine Darstellung von BCD-Ziffern siehe Kap. 3.6

Condition Code Register:

X	N	Z	V	C
*	?	*	?	*

C wird gesetzt, wenn eine (dezimale) "Leihe" generiert wird. Wird sonst zurückgesetzt.
V nicht definiert
Z wird gelöscht, wenn das Ergebnis ungleich Null ist. Bleibt sonst unverändert.
N nicht definiert
X Erhält den gleichen Wert wie das C-Bit.



Negiere BCD-Zahl mit Extend-Bit
negate decimal with extend

Format des Befehlswortes:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	0	0	Effektive Adresse					
										Mode			Register		

Bit 5..0 Die Effektive Adresse wählt den Zieloperand an. Die folgenden Adressierungsarten sind erlaubt:

Adr.-Art	Mode	Reg
Dn	000	R:Dn
An	nicht erlaubt	
(An)	010	R:An
(An)+	011	R:An
-(An)	100	R:An
d16(An)	101	R:An
d8(An,Xi)	110	R:An

Adr.-Art	Mode	Reg
xxx.W	111	000
xxx.L	111	001
d16(PC)	nicht erlaubt	
d8(PC,Xi)	nicht erlaubt	
#<data>	nicht erlaubt	
Erläuterung siehe Kapitel 5		

PROGRAMMIERHINWEIS:

Mit den Befehlen ABCD, NBCD und SBCD können Sie BCD-Zahlen mit beliebig vielen Stellen automatisch abarbeiten.

Legen Sie dazu - in Übereinstimmung mit der 68000-Architektur - die niederwertigen Bytes der Binärzahl auf die höhere Adresse, und die hochwertigen Bytes auf die niedrige Adresse ab. Ein Byte enthält zwei BCD-Zahlen.

Negiere BCD-Zahl mit Extend-Bit
negate decimal with extend

NBCD

Vor Anfang der Operation setzen Sie das Z-Bit und löschen Sie das X-Bit. Gebrauchen Sie dazu z.B. den Befehl MOVE.W #4, CCR.

Sie fangen die Abarbeitung bei dem niederwertigen Byte auf der höchsten Adresse an, und benutzen die prädekrement-Adressierung -(An), um nacheinander die hochwertigeren Bytes auf der niedrig werdenden Adresse zu adressieren.

Wenn bei der Verarbeitung eines Bytes ein "Leihen" bzw. ein Übertrag entsteht, wird das X-Bit gesetzt. Dieses Leihen bzw. dieser Übertrag wird rechnerisch korrekt verarbeitet, indem das nächsthöhere Byte um eins (1) verringert wird. Das X-Bit übernimmt im nächsten Aufruf diese Aufgabe.

Das Z-Bit wird zurückgesetzt, wenn ein Byte ungleich Null ist. Es ändert sich aber nicht, wenn ein Byte Null ist. Ist also am Ende der Operation das Z-Bit immer noch gesetzt, ist die ganze Binärzahl Null.

Siehe auch: ABCD, NEGX, SBCD

NEG

Negiere Operand
negate

0 - Ziel -> Ziel

Operandgröße: NEG.B Byte (8 Bit)
NEG.W Wort (16 Bit)
NEG.L Langwort (32 Bit)

Assembler Syntax: NEG.x <ea>
(x entspricht B, W, L)

Beschreibung:

Der Operand der Zieladresse <ea> wird von Null subrahiert. Das Ergebnis wird im Zieloperand <ea> abgespeichert. Nach der Operation ist der Operand durch sein Zweierkomplement ersetzt worden.

Die Operandgröße ist Byte, Wort oder Langwort.

Condition Code Register:

X	N	Z	V	C
*	*	*	*	*

C wird gesetzt, wenn ein "Leihen" generiert wird.
Wird sonst zurückgesetzt.
V wird gesetzt, wenn ein Überlauf generiert wird.
Wird sonst zurückgesetzt.
Z wird gesetzt, wenn das Ergebnis gleich Null ist.
Wird sonst zurückgesetzt.
N wird gesetzt, wenn das Ergebnis negativ ist.
Wird sonst zurückgesetzt.
X Erhält den gleichen Wert wie das C-Bit.

Negiere Operand
negate

NEG

Format des Befehlswortes:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	Größe		Effektive Adresse					
										Mode			Register		

Bit 7..6 Größe-Feld:
 00 Byte-Befehl NEG.B
 01 Wort-Befehl NEG.W
 10 Langwort-Befehl NEG.L

Bit 5..0 Die Effektive Adresse wählt den Zieloperand an. Die folgenden Adressierungsarten sind erlaubt:

Adr.-Art	Mode	Reg
Dn	000	R:Dn
An	nicht erlaubt	
(An)	010	R:An
(An)+	011	R:An
-(An)	100	R:An
d16(An)	101	R:An
d8(An,Xi)	110	R:An

Adr.-Art	Mode	Reg
xxx.W	111	000
xxx.L	111	001
d16(PC)	nicht erlaubt	
d8(PC,Xi)	nicht erlaubt	
#<data>	nicht erlaubt	
Erläuterung siehe Kapitel 5		

Benutzen Sie

NOT: wenn Sie das Einerkomplement haben möchten (Bitweise Komplement)

NEGX: wenn sie über das X-Bit den Operand um eins vergrößern möchten (Abarbeitung von Reihen)

Siehe auch: NEGX, EXT



Negiere Operand mit Extend-Bit
negate with extend

0 - Ziel - X -> Ziel

Operandgröße: NEGX.B Byte (8 Bit)
NEGX.W Wort (16 Bit)
NEGX.L Langwort (32 Bit)

Assembler Syntax: NEGX.x <ea>
(x entspricht B, W, L)

Beschreibung:

Der Operand der Zieladresse <ea> und das Extend-Bit werden von Null subrahiert. Das Ergebnis wird im Zieloperand <ea> abgespeichert.

Die Operandgröße ist Byte, Wort oder Langwort.

X	N	Z	V	C
*	*	*	*	*

Condition Code Register:

C wird gesetzt, wenn ein "Leihen" generiert wird. Wird sonst zurückgesetzt.
V wird gesetzt, wenn ein Überlauf generiert wird. Wird sonst zurückgesetzt.
Z wird gelöscht, wenn das Ergebnis ungleich Null ist. Bleibt sonst unverändert.
N wird gesetzt, wenn das Ergebnis negativ ist. Wird sonst zurückgesetzt.
X Erhält den gleichen Wert wie das C-Bit.

Negiere Operand mit Extend-Bit
negate with extend

NEGX

Format des Befehlswortes:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	0	Größe		Effektive Adresse					
										Mode			Register		

Bit 7..6 Größe-Feld:
00 Byte-Befehl NEGX.B
01 Wort-Befehl NEGX.W
10 Langwort-Befehl NEGX.L

Bit 5..0 Die Effektive Adresse wählt den Zieloperand an. Die folgenden Adressierungsarten sind erlaubt:

Adr.-Art	Mode	Reg
Dn	000	R:Dn
An	nicht erlaubt	
(An)	010	R:An
(An)+	011	R:An
-(An)	100	R:An
d16(An)	101	R:An
d8(An,Xi)	110	R:An

Adr.-Art	Mode	Reg
xxx.W	111	000
xxx.L	111	001
d16(PC)	nicht erlaubt	
d8(PC,Xi)	nicht erlaubt	
#<data>	nicht erlaubt	
Erläuterung siehe Kapitel 5		

PROGRAMMIERHINWEIS:

Mit den Befehlen NEGX, ADDX, SUBX können Sie Binärzahlen mit beliebig vielen Stellen automatisch abarbeiten.

Legen Sie dazu - in Übereinstimmung mit der 68000-Architektur - die niederwertigen Teile der Binärzahl auf die höhere Adresse, und die hochwertigen Teile auf die niedrige Adresse ab. Diese Teile können - je nach Ihrer Wahl - Langwörter, Wörter oder Bytes sein.

NEGX

Negiere Operand mit Extend-Bit
negate with extend

Vor Anfang der Operation setzen Sie das Z-Bit und löschen Sie das X-Bit. Gebrauchen Sie dazu z.B. den Befehl MOVE.W #4, CCR.

Sie fangen die Abarbeitung bei dem niederwertigen Teil auf der höchsten Adresse an, und benutzen die prädekrement-Adressierung -(An), um nacheinander die hochwertigeren Teile auf der niedrig werdenden Adresse zu adressieren.

Wenn bei der Verarbeitung eines Teils ein "Leihen" bzw. ein Übertrag entsteht, wird das X-Bit gesetzt. Dieses Leihen bzw. dieser Übertrag wird rechnerisch korrekt verarbeitet, indem das nächsthöhere Teil um eins (1) verringert wird. Das X-Bit übernimmt im nächsten Aufruf diese Aufgabe.

Das Z-Bit wird zurückgesetzt, wenn ein Teil ungleich Null ist. Es ändert sich aber nicht, wenn ein Teil Null ist. Ist also am Ende der Operation das Z-Bit immer noch gesetzt, ist die ganze Binärzahl Null.

Siehe auch: NEG, EXT

Tue Nichts
no Operation

NOP

Operation: keine

Operandgröße: keine

Assembler Syntax: NOP

Beschreibung:

Es findet keine Aktion statt.
Natürlich läuft der Programmzähler - wie bei jedem anderen Befehl auch - weiter bis zum nächsten Befehl.

Bei der Programm-Entwicklung mit dem Debugger kann der Befehl NOP benutzt werden, um andere Befehle wegzunehmen, oder, um als Platzhalter für zukünftige Befehle zu dienen.

Bei laufzeitkritischen Programmen können NOPs für eine Feinjustage der Laufzeit benutzt werden. (Längere Verzögerungen erhält man mit Zählschleifen oder durch Ausnutzung der Clock-Interrupts.)

X	N	Z	V	C
-	-	-	-	-

Condition Code Register:
bleibt unverändert

Format des Befehlswortes:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	0	0	0	1

($4E71)

NOT

Logisches Komplement
logical complement

~Ziel -> Ziel

Operandgröße: NOT.B Byte (8 Bit)
NOT.W Wort (16 Bit)
NOT.L Langwort (32 Bit)

Assembler Syntax: NOT.x <ea>
(x entspricht B, W, L)

Beschreibung:

Vom Zieloperand wird das Einerkomplement genommen und im Zieloperand abgespeichert.

Das Einerkomplement wird auch als "NICHT-Verknüpfung" bezeichnet.

> Zur Erinnerung die "Nicht-Verknüpfungen":
> ~0 = 1
> ~1 = 0
> Das bedeutet konkret, das jedes gesetztes Bit zurückgesetzt, und jedes zurückgesetztes Bit gesetzt wird.

Condition Code Register:

X	N	Z	V	C
-	*	*	0	0

C wird zurückgesetzt.
V wird zurückgesetzt.
Z wird gesetzt, wenn das Ergebnis gleich Null ist. Wird sonst zurückgesetzt.
N wird gesetzt, wenn das hochwertige Bit des Ergebnisses gesetzt ist (zeigt ein negatives Ergebnis). Wird sonst zurückgesetzt.
X bleibt unverändert.

Logisches Komplement
logical complement

NOT

Format des Befehlswortes:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	1	0	Größe		Effektive Adresse					
										Mode			Register		

Bit 7..6 Größe-Feld:
00 Byte-Befehl NOT.B
01 Wort-Befehl NOT.W
10 Langwort-Befehl NOT.L

Aufbau der Argumentwörter siehe Kap. 3.8

Bit 5..0 Die Effektive Adresse wählt den Zieloperand an. Die folgenden Adressierungsarten sind erlaubt:

Adr.-Art	Mode	Reg
Dn	000	R:Dn
An	nicht erlaubt	
(An)	010	R:An
(An)+	011	R:An
-(An)	100	R:An
d16(An)	101	R:An
d8(An,Xi)	110	R:An

Adr.-Art	Mode	Reg
xxx.W	111	000
xxx.L	111	001
d16(PC)	nicht erlaubt	
d8(PC,Xi)	nicht erlaubt	
#<data>	nicht erlaubt	
Erläuterung siehe Kapitel 5		

Benutzen Sie
NEG:
wenn Sie das Zweierkomplement des Operanden suchen.

Siehe auch: OR, EOR, NOT, TST



Logisches ODER
inclusive or logical

Quelle v Ziel -> Ziel

Operandgröße: OR.B Byte (8 Bit)
OR.W Wort (16 Bit)
OR.L Langwort (32 Bit)

Assembler Syntax:
OR.x <ea>, Dn
OR.x Dn, <ea>
(x entspricht B, W, L)

Operation:
<ea> v Dn -> Dn
Dn v <ea> -> <ea>

Beschreibung:

Der Quelloperand wird mit dem Zieloperand bitweise ODER-verknüpft, und das Ergebnis wird im Zieloperand abgespeichert.

Zur Erinnerung die ODER-Verknüpfungen:
0 v 0 = 0
0 v 1 = 1
1 v 0 = 1
1 v 1 = 1

Das Ergebnisbit wird gesetzt, wenn das eine oder das andere Eingangsbit gesetzt sind.

Einer der beiden Operanden muß ein Datenregister sein.

Die Größe des Operanden sowie die Angabe, welcher Operand das Datenregister ist, sind im Mode-Feld enthalten.

Logisches ODER
inclusive or logical

OR

Condition Code Register:

```
  X   N   Z   V   C
|   |   |   |   |   |
| - | * | * | 0 | 0 |
|___|___|___|___|___|
```

C wird zurückgesetzt.
V wird zurückgesetzt.
Z wird gesetzt, wenn das Ergebnis gleich Null ist.
Wird sonst zurückgesetzt.
N wird gesetzt, wenn das hochwertige Bit des Ergebnisses gesetzt ist (zeigt ein negatives Ergebnis).
Wird sonst zurückgesetzt.
X bleibt unverändert.

Format des Befehlswortes:

```
| 15  14  13  12| 11  10  9   8 | 7   6   5   4 | 3   2   1   0 |
|   |   |   |   |             |            |                    |
| 1 | 0 | 0 | 0 |   Daten-    |Operations-|    Effektive Adresse |
|___|___|___|___|  Register   |   Mode|   |  Mode      |  Register |
```

Bit 11..9 Registerfeld: wählt eines der acht Datenregister an.

Bit 8..6 Feld Operationsmode:

OR.B	OR.W	OR.L	Operation
000	001	010	<ea> v Dn -> Dn
100	101	110	Dn v <ea> -> <ea>

OR

Logisches ODER
inclusive or logical

Bit 5..0 Wenn die Effektive Adresse der **Quelloperand** ist (also <ea> v Dn -> Dn), sind die folgenden Adressierungsarten erlaubt:

Adr.-Art	Mode	Reg
Dn	000	R:Dn
An	nicht erlaubt	
(An)	010	R:An
(An}+	011	R:An
-(An)	100	R:An
d16(An)	101	R:An
d8(An,Xi)	110	R:An

Adr.-Art	Mode	Reg
xxx.W	111	000
xxx.L	111	001
d16(PC)	111	010
d8(PC,Xi)	111	011
#<data>	111	100
Erläuterung siehe Kapitel 5		

Bit 5..0 Wenn die Effektive Adresse der **Zieloperand** ist (also Dn v <ea> -> <ea>), sind die folgenden Adressierungsarten erlaubt:

Adr.-Art	Mode	Reg
Dn	nicht erlaubt	
An	nicht erlaubt	
(An)	010	R:An
(An)+	011	R:An
-(An)	100	R:An
d16(An)	101	R:An
d8(An,Xi)	110	R:An

Adr.-Art	Mode	Reg
xxx.W	111	000
xxx.L	111	001
d16(PC)	nicht erlaubt	
d8(PC,Xi)	nicht erlaubt	
#<data>	nicht erlaubt	
Erläuterung siehe Kapitel 5		

HINWEIS:
Wenn der Zieloperand ein Datenregister sein soll, kann er nicht mit der Adressierungsart <ea> angewählt werden, sondern nur mit der Adressierungsart Dn.

Logisches ODER
inclusive or logical

OR

Benutzen Sie

ORI:	wenn einer der Operanden eine Konstante ist.

Siehe auch: AND, EOR, NOT, TST



ODER mit Konstante
inclusive or immediate

Konstante v Ziel -> Ziel

Operandgröße: ORI.B Byte (8 Bit)
ORI.W Wort (16 Bit)
ORI.L Langwort (32 Bit)

Assembler Syntax: ORI.x #<data>, <ea> (Quelle, Ziel)
(x entspricht Bf W, L)

Beschreibung:
Die Konstante, die im Speicher unmittelbar dem Befehlswort folgt, wird mit dem Zieloperand <ea> bitweise ODER-verknüpft. Das Ergebnis wird im Zieloperand <ea> abgespeichert.

Die Größe der Konstante entspricht der Operandgröße.

Condition Code Register:

X	N	Z	V	C
-	*	*	0	0

C wird zurückgesetzt.
V wird zurückgesetzt.
Z wird gesetzt, wenn das Ergebnis gleich Null ist. Wird sonst zurückgesetzt.
N wird gesetzt, wenn das hochwertige Bit des Ergebnisses gesetzt ist (zeigt ein negatives Ergebnis).
Wird sonst zurückgesetzt.
X bleibt unverändert.

ODER mit Konstante
inclusive or immediate

ORI

Format des Befehlswortes:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	Größe		Effektive Adresse Mode			Register		
1. Argumentwort: Wort Daten								bzw. Byte Daten							
2. Argumentwort: Langwort-Daten (einschließlich voriges Wort)															

Bit 7..6 Größe-Feld:
00 Byte-Befehl ORI.B
01 Wort-Befehl ORI.W
10 Langwort-Befehl ORI.L

Aufbau der Argumentwörter siehe Kap. 3.8

Bit 5..0 Die Effektive Adresse wählt den Zieloperand an. Die folgenden Adressierungsarten sind erlaubt:

Adr.-Art	Mode	Reg
Dn	000	R:Dn
An	nicht erlaubt	
(An)+	011	R:An
(An)	010	R:An
-(An)	100	R:An
d16(An)	101	R:An
d8(An,Xi)	110	R:An

Adr.-Art	Mode	Reg
xxx.W	111	000
xxx.L	111	001
d16(PC)	nicht erlaubt	
d8(PC,Xi)	nicht erlaubt	
#<data>	nicht erlaubt	
Erläuterung siehe Kapitel 5		

Argumentwort:
Bit 7..6 = 00 -> Datenfeld ist die niederwertige Hälfte des 1. Argumentwortes
Bit 7..6 = 01 -> Datenfeld ist das 1. Argumentwort
Bit 7..6 = 10 -> Datenfeld ist 1. + 2. Argumentwort

Siehe auch: ANDI, OR, EORI, NOT, TST

ORI zum CCR

ODER mit Konstante zum CCR
inclusive or immediate to CCR

Konstante v CCR -> CCR

Operandgröße: Byte (8 Bit)

Assembler Syntax: ORI #<data>, CCR (Quelle, Ziel)

Beschreibung:

Die Konstante, die im Speicher unmittelbar dem Befehlswort folgt, wird mit dem Condition Code Register bitweise ODER-verknüpft. Das Ergebnis wird im Condition Code Register abgespeichert.

Die Operandgröße ist ein Byte.

Condition Code Register:

X	N	Z	V	C
*	*	*	*	*

C wird gesetzt, wenn Bit 0 der Konstante gesetzt ist. Bleibt sonst unverändert.
V wird gesetzt, wenn Bit 1 der Konstante gesetzt ist. Bleibt sonst unverändert.
Z wird gesetzt, wenn Bit 2 der Konstante gesetzt ist. Bleibt sonst unverändert.
N wird gesetzt, wenn Bit 3 der Konstante gesetzt ist. Bleibt sonst unverändert.
X wird gesetzt, wenn Bit 4 der Konstante gesetzt ist. Bleibt sonst unverändert.

ODER mit Konstante zum CCR
inclusive or immediate to CCR

ORI zum CCR

Format des Befehlswortes:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	0
0	0	0	0	0	0	0	0	Konstante (8 Bit)							

Benutzen Sie

ORI zum SR:	wenn Sie auch das System Byte ODER-verknüpfen möchten.
ANDI zum CCR:	wenn Sie das CCR UND-verknüpfen möchten.
EORI zum CCR:	wenn Sie das CCR Exclusiv-ODER-verknüpfen möchten.
MOVE zum CCR:	wenn Sie das CCR ohne Rücksicht auf die bestehenden Bits ändern möchten.

Siehe auch:
ANDI zum SR, ORI zum SR

ORI zum SR

ODER mit Konstante zum SR
inclusive or immediate to SR
(privilegierter Befehl)

Wenn Supervisor Mode: Konstante v SR -> SR

Wenn User Mode: **Auslösung Exception 8 (Kap 6)**
(Verletzung Privilegium)

Operandgröße: Wort (16 Bit)

Assembler Syntax: ORI #<data>, SR (Quelle, Ziel)

Beschreibung:

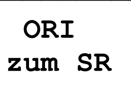
Wenn der Prozessor sich im Supervisor Mode befindet, wird die Konstante, die im Speicher unmittelbar dem Befehlswort folgt, mit dem Status Register bitweise ODER-verknüpft. Das Ergebnis wird im Status Register abgespeichert.

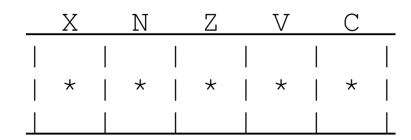
Wenn der Prozessor dagegen im User Mode ist, wird eine Exception ausgelöst.

Das Status Register wird in Kap. 2 beschrieben.

Die Operandgröße ist ein Wort (16 Bit).

ODER mit Konstante zum SR
inclusive or immediate to SR
(privilegierter Befehl)


ORI
zum SR

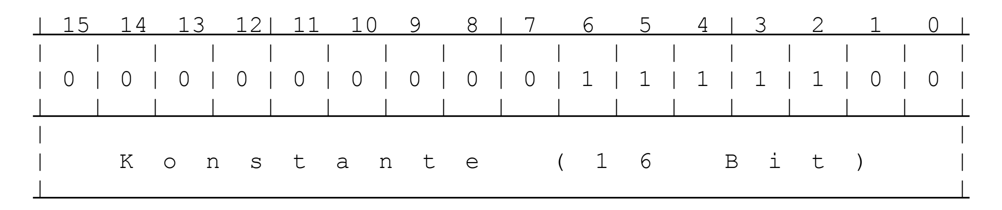


X	N	Z	V	C
*	*	*	*	*

Condition Code Register:

C wird gesetzt, wenn Bit 0 der Konstante gesetzt ist. Bleibt sonst unverändert.
V wird gesetzt, wenn Bit 1 der Konstante gesetzt ist. Bleibt sonst unverändert.
Z wird gesetzt, wenn Bit 2 der Konstante gesetzt ist. Bleibt sonst unverändert.
N wird gesetzt, wenn Bit 3 der Konstante gesetzt ist. Bleibt sonst unverändert.
X wird gesetzt, wenn Bit 4 der Konstante gesetzt ist. Bleibt sonst unverändert.

Format des Befehlswortes:



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	1	1	1	1	1	0	0
K o n s t a n t e (1 6 B i t)															

Benutzen Sie

ANDI zum SR: wenn Sie das SR UND-verknüpfen möchten.

MOVE zum SR: wenn Sie das SR ohne Rücksicht auf die bestehenden Bits ändern möchten.

Siehe auch: ANDI zum SR, ORI zum CCR, EORI zum CCR

PEA

Lege effektive Adresse im Stack ab
push effective address

SP - 4 -> SP; <ea> -> (SP)

Operandgröße: Langwort (32 Bit)

Assembler Syntax: PEA <ea>

Beschreibung:

Die effektive Adresse wird ermittelt und auf den Stack gepopped. Die Datengröße ist ein Langwort.

X	N	Z	V	C
-	-	-	-	-

Condition Code Register:
bleibt unverändert

Format des Befehlswortes:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	0	1	Effektive Adresse					
										Mode			Register		

Bit 5..0 spezifiziert die Quelladresse der tion, die in den Stack kopiert werden soll.

Adr.-Art	Mode	Reg
Dn	nicht erlaubt	
An	nicht erlaubt	
(An)	010	R:An
(An)+	nicht erlaubt	
-(An)	nicht erlaubt	
d16(An)	101	R:An
d8(An,Xi)	110	R:An

Adr.-Art	Mode	Reg
xxx.W	111	000
xxx.L	111	001
d16(PC)	111	010
d8(PC,Xi)	111	011
#<data>	nicht erlaubt	
Erläuterung siehe Kapitel 5		

Rücksetzen Eingabe-Ausgabe
reset external devices
(privilegierter Befehl)

RESET

Wenn Supervisor Mode:
Rücksetzen der Eingabe-Ausgabe-Kanäle
Wenn User Mode:
Exception 8 wird ausgelöst (siehe Kap. 6)
(Verletzung Privilegium)

Operandgröße: keine

Assembler Syntax: RESET

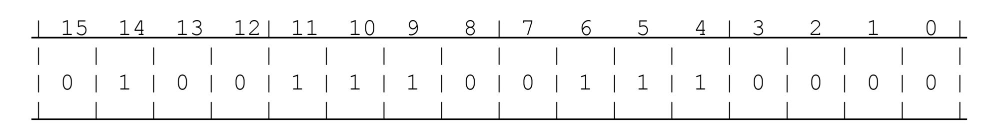
Beschreibung:

Alle Eingabe-Ausgabe-Bausteine, die mit der Leitung verbunden sind, werden zurückgesetzt.

X	N	Z	V	C
-	-	-	-	-

Condition Code Register:
bleibt unverändert

Format des Befehlswortes:



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	0	0	0	0

($4E70)

Verwechseln Sie diesen Befehl nicht mit dem E x c e p t i o n Reset. Der Exception Reset setzt den Programmzähler neu und ist für den Systemstart gedacht.

ROL

Rotiere links ohne Extend-Bit
rotate left without extend

Ziel rotiert durch <Zahl> -> Ziel

Operandgröße: ROL.B Byte (8 Bit)
ROL.W Wort (16 Bit)
ROL.L Langwort (32 Bit)
ROL Wort (16 Bit)

Assembler Syntax: ROL.x Dn, Dn (Quelle, Ziel)
ROL.x #<data>, Dn
(x entspricht B, W, L)
ROL <ea>

Beschreibung:
Die Bits des Operanden werden nach links rotiert. Bei jedem Rotationsschritt passiert folgendes:

o Das C-Bit erhält den Wert des hochwertigen Bits. Der Wert wird außerdem zwischengespeichert.

o Danach erhält das hochwertige Bit den Wert des Bits rechts daneben. Dieses Bit erhält dann den Wert seines rechten Nachbarns usw, bis Bit 1 den Wert von Bit 0 erhält.

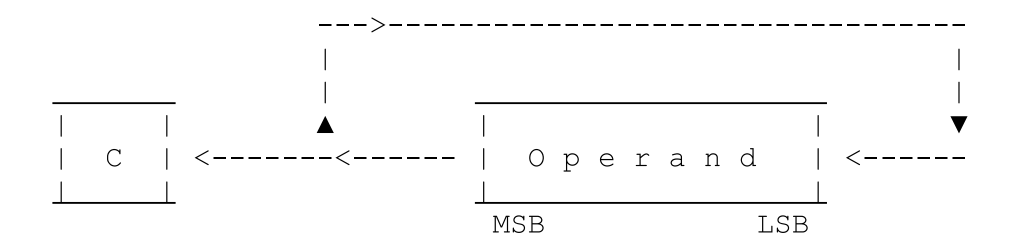
o Zum Schluß erhält Bit 0 den Wert, der vorher im hochwertigen Bit war.

Das X-Bit ist von der Rotation nicht betroffen.

Für die Gesamtzahl der Rotationsschritte siehe auf der nächsten Seite.

Rotiere links ohne Extend-Bit
rotate left without extend

ROL



Es gibt drei Befehlsformen.

o Der Befehl
ROL.x #<data>, Dn (x entspricht B, W, L)
rotiert ein Datenregister nach links um soviele Positionen, wie in der Konstante angegeben ist. Die Maximalzahl ist acht.

o Der Befehl
ROL.x Dn, Dn (x entspricht B, W, L)
rotiert ein Datenregister nach links. Ein zweites Datenregister legt fest, um wieviele Positionen rotiert wird.

o Der Befehl
ROL <ea>
rotiert eine Speicherstelle (16 Bit) um eine Position nach links.

ROL

Rotiere links ohne Extend-Bit
rotate left without extend

Condition Code Register:

X	N	Z	V	C
-	*	*	0	*

C erhält den Wert, der zuletzt aus dem hochwertigen Bit des Operanden herausgeschoben wurde. Wird zurückgesetzt beim Rotieren um Null Positionen.
V wird zurückgesetzt.
Z wird gesetzt, wenn das Ergebnis gleich Null ist. Wird sonst zurückgesetzt.
N wird gesetzt, wenn das hochwertige Bit des Ergebnisses gesetzt ist. Wird sonst zurückgesetzt.
X ändert sich nicht.

Rotiere links ohne Extend-Bit
rotate left without extend

ROL

Assembler Syntax: ROL.x Dn, Dn
ROL.x #<data>, Dn
(x entspricht B, W, L)

Dazu gehört das folgende Format des Befehlswortes:

15	14	13	12	11 10 9	8	7 6	5	4	3	2 1 0
1	1	1	0	Zähl-Register	1	Größe	i	1	1	Daten-Register

Bit 11..9 Zählregister-Feld
Wenn i = 0:
Die Bits 11..9 geben an, um wieviele Positionen die Bits des Zieloperanden nach links rotiert werden. Dabei entspricht 001 einer Position usw., bis 111 sieben Positionen entspricht. 000 entspricht aber acht Positionen. (Konstante)
Wenn i = 1:
Die Bits 11..9 wählen ein Datenregister Dn an. Die niederwertigen 6 Bits des Datenregisters Dn geben an, um wieviele Positionen die Bits des Zieloperanden nach links rotiert werden.

Bit 7..6 Größe-Feld
00 Byte-Befehl ROL.B
01 Wort-Befehl ROL.W
10 Langwort-Befehl ROL.W

Bit 5 i-Feld
0 Die Bits 11..9 beziehen sich auf eine Konstante
1 Die Bits 11..9 beziehen sich auf ein Datenregister.

Bit 2..0 Register-Feld
wählt ein Datenregister als Zieloperand an.



Rotiere links ohne Extend-Bit
rotate left without extend

Assembler Syntax: ROL <ea>

Die 16 Bits des angewählten Speicherwortes werden um eine Position nach links rotiert.

Dazu gehört das folgende Format des Befehlswortes:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	1	1	1	1	Effektive Adresse					
										Mode			Register		

Bit 5..0 Wählt die Effektive Adresse des Quell-operanden an. Die folgenden Adressierungsarten sind erlaubt:

Adr.-Art	Mode	Reg
Dn	nicht erlaubt	
An	nicht erlaubt	
(An)	010	R:An
(An)+	011	R:An
-(An)	100	R:An
d16(An)	101	R:An
d8(An,Xi)	110	R:An

Adr.-Art	Mode	Reg
xxx.W	111	000
xxx.L	111	001
d16(PC)	nicht erlaubt	
d8(PC,Xi)	nicht erlaubt	
#<data>	nicht erlaubt	
Erläuterung siehe Kapitel 5		

Für die Adressierungsart Dn siehe auf der vorherigen Seite.

Benutzen Sie
ROR: wenn nach rechts rotiert werden soll.
ROXL: Wenn die Rotation auch durch das X-Bit laufen soll.

Siehe auch: ASL, ASR, LSL, LSR, ROXR, SWAP

Rotiere rechts ohne Extend-Bit
rotate right without extend

ROR

Ziel rotiert durch <Zahl> -> Ziel

```
Operandgröße:  ROR.B  Byte (8 Bit)
               ROR.W  Wort (16 Bit)
               ROR.L  Langwort (32 Bit)
               ROR  Wort (16 Bit)

Assembler      ROR.x  Dn, Dn  (Quelle, Ziel)
Syntax:        ROR.x  #<data>, Dn
                  (x entspricht B, W, L)
               ROR    <ea>
```

Beschreibung:
Die Bits des Operanden werden nach rechts rotiert.
Bei jedem Rotationsschritt passiert folgendes:

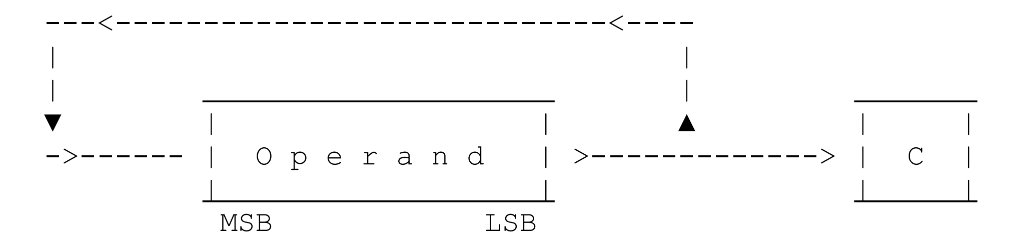
- Bit 0 gibt seinen Wert an das C-Bit ab. Der Wert wird außerdem zwischengespeichert.
- Danach gibt Bit 1 seinen Wert an Bit 0, und Bit 2 seinen Wert an Bit 1 usw., bis das hochwertige Bit seinen Wert an seinen rechten Nachbarn abgibt.
- Zum Schluß erhält das hochwertige Bit den Wert, der vorher in Bit 0 war.

Das X-Bit ist von der Rotation nicht betroffen.

Für die Gesamtzahl der Rotationsschritte siehe auf der nächsten Seite.

ROR

Rotiere rechts ohne Extend-Bit
rotate right without extend



Es gibt drei Befehlsformen.

o Der Befehl
ROR.x #<data>, Dn (x entspricht B, W, L)
rotiert ein Datenregister nach rechts um soviele Positionen, wie in der Konstante angegeben ist. Die Maximalzahl ist acht.

o Der Befehl
ROR.x Dn, Dn (x entspricht B, W, L)
rotiert ein Datenregister nach rechts. Ein zweites Datenregister legt fest, um wieviele Positionen rotiert wird.

o Der Befehl
ROR <ea>
rotiert eine Speicherstelle (16 Bit) um eine Position nach rechts.

Rotiere rechts ohne Extend-Bit
rotate right without extend

ROR

Condition Code Register:

X	N	Z	V	C
-	*	*	0	*

C erhält den Wert, der zuletzt aus dem hochwertigen Bit des Operanden herausgeschoben wurde. Wird zurückgesetzt beim Rotieren um Null Positionen.
V wird zurückgesetzt.
Z wird gesetzt, wenn das Ergebnis gleich Null ist. Wird sonst zurückgesetzt.
N wird gesetzt, wenn das hochwertige Bit des Ergebnisses gesetzt ist. Wird sonst zurückgesetzt.
X ändert sich nicht.

ROR

Rotiere rechts ohne Extend-Bit
rotate right without extend

Assembler Syntax: ROR.x Dn, Dn
ROR.x #<data>, Dn
(x entspricht B, W, L)

Dazu gehört das folgende Format des Befehlswortes:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	Zähl-Register			0	Größe		i	1	1	Daten-Register		

Bit 11..9 Zählregister-Feld
Wenn i = 0:
Die Bits 11..9 geben an, um wieviele Positionen die Bits des Zieloperanden nach rechts rotiert werden. Dabei entspricht 001 einer Position usw., bis 111 sieben Positionen entspricht. 000 entspricht aber acht Positionen. (Konstante)
Wenn i = l:
Die Bits 11..9 wählen ein Datenregister Dn an. Die niederwertigen 6 Bits des Datenregisters Dn geben an, um wieviele Positionen die Bits des Zieloperanden nach rechts rotiert werden.

Bit 7..6 Größe-Feld
00 Byte-Befehl ROR.B
01 Wort-Befehl ROR.W
10 Langwort-Befehl ROR.W

Bit 5 i-Feld
0 Die Bits 11..9 beziehen sich auf eine Konstante
l Die Bits 11..9 beziehen sich auf ein Datenregister.

Bit 2..0 Register-Feld
wählt ein Datenregister als Zieloperand an.

Rotiere rechts ohne Extend-Bit
rotate right without extend **ROR**

Assembler Syntax: ROR <ea>

Die 16 Bits des angewählten Speicherwortes werden um eine Position nach rechts rotiert.

Dazu gehört das folgende Format des Befehlswortes:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	1	0	1	1	Effektive Adresse					
										Mode			Register		

Bit 5..0 Wählt die Effektive Adresse des Quell-operanden an. Die folgenden Adressierungsarten sind erlaubt:

Adr.-Art	Mode	Reg
Dn	nicht erlaubt	
An	nicht erlaubt	
(An)	010	R:An
(An)+	011	R:An
-(An)	100	R:An
d16(An)	101	R:An
d8(An,Xi)	110	R:An

Adr.-Art	Mode	Reg
xxx. W	111	000
xxx.L	111	001
d16(PC)	nicht erlaubt	
d8(PC,Xi)	nicht erlaubt	
#<data>	nicht erlaubt	
Erläuterung siehe Kapitel 5		

Für die Adressierungsart Dn siehe auf der vorherigen Seite.

Benutzen Sie
ROL: wenn nach links rotiert werden soll.
ROXR: Wenn die Rotation auch durch das X-Bit laufen soll.

Siehe auch: ASL, ASR, LSL, LSR, ROXL, SWAP



Rotiere links mit Extend-Bit
rotate left with extend

```
Ziel rotiert durch <Zahl> -> Ziel

Operandgröße:  ROXL.B  Byte (8 Bit)
               ROXL.W  Wort (16 Bit)
               ROXL.L  Langwort (32 Bit)
               ROXL  Wort (16 Bit)

Assembler      ROXL.x  Dn, Dn          (Quelle, Ziel)
Syntax:        ROXL.x  #<data>, Dn
                  (x entspricht B, W, L)
               ROXL    <ea>
```

Beschreibung:

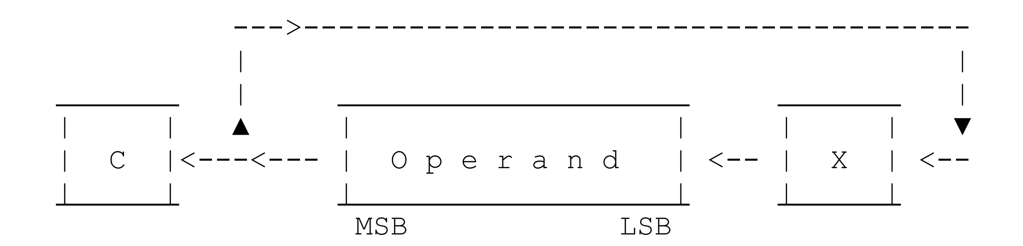
Die Bits des Operanden werden nach links rotiert. Bei jedem Rotationsschritt passiert folgendes:

- o Das C-Bit erhält den Wert des hochwertigen Bits. Der Wert wird außerdem zwischengespeichert..
- o Danach bekommt das hochwertige Bit den Wert des Bits rechts daneben. Dieses Bit wiederum den Wert seines rechten Nachbarns usw, bis Bit 1 den Wert von Bit 0 erhält.
- o Dann erhält Bit 0 den Wert des X-Registers.
- o Zum Schluß erhält das X-Register den Wert, der vorher im hochwertigen Bit war.

Für die Gesamtzahl der Rotationsschritte siehe weiter unten.

Rotiere links mit Extend-Bit
rotate left with extend

ROXL



Es gibt drei Befehlsformen.

- Der Befehl
  ROXL.x #<data>, Dn (x entspricht B, W, L)
  rotiert ein Datenregister nach links um soviele Positionen, wie in der Konstante angegeben ist. Die Maximalzahl ist acht.

- Der Befehl
  ROXL.x Dn, Dn (x entspricht B, W, L)
  rotiert ein Datenregister nach links. Ein zweites Datenregister legt fest, um wieviele Positionen rotiert wird.

- Der Befehl
  ROXL <ea>
  rotiert eine Speicherstelle (16 Bit) um eine Position nach links.

ROXL

Rotiere links mit Extend-Bit
rotate left with extend

Condition Code Register:

X	N	Z	V	C
-	*	*	0	*

C erhält den Wert, der zuletzt aus dem hochwertigen Bit des Operanden herausgeschoben wurde. Wird zurückgesetzt beim Rotieren um Null Positionen.
V wird zurückgesetzt.
Z wird gesetzt, wenn das Ergebnis gleich Null ist. Wird sonst zurückgesetzt.
N wird gesetzt, wenn das hochwertige Bit des Ergebnisses gesetzt ist. Wird sonst zurückgesetzt.
X erhält den Wert, der zuletzt aus dem hochwertigen Bit des Operanden herausgeschoben wurde. Bleibt unverändert bei Schieben um Null Positionen.

Rotiere links mit Extend-Bit
rotate left with extend

ROXL

Assembler Syntax: ROXL.x Dn, Dn
ROXL.x #<data>, Dn
(x entspricht B, W, L)

Dazu gehört das folgende Format des Befehlswortes:

15	14	13	12	11 10 9	8	7 6	5	4	3	2 1 0
1	1	1	0	Zähl-Register	1	Größe	i	1	0	Daten-Register

Bit 11..9 Zählregister-Feld
Wenn i = 0:
Die Bits 11..9 geben an, um wieviele Positionen die Bits des Zieloperanden nach links rotiert werden. Dabei entspricht 001 einer Position usw., bis 111 sieben Positionen entspricht. 000 entspricht aber acht Positionen. (Konstante)
Wenn i = l:
Die Bits 11..9 wählen ein Datenregister Dn an. Die niederwertigen 6 Bits des Datenregisters Dn geben an, um wieviele Positionen die Bits des Zieloperanden nach links rotiert werden.

Bit 7..6 Größe-Feld
00 Byte-Befehl ROXL.B
01 Wort-Befehl ROXL.W
10 Langwort-Befehl ROXL.W

Bit 5 i-Feld
0 Die Bits 11..9 beziehen sich auf eine Konstante
l Die Bits 11..9 beziehen sich auf ein Datenregister.

Bit 2..0 Register-Feld
wählt ein Datenregister als Zieloperand an.

ROXL

Rotiere links mit Extend-Bit
rotate left with extend

Assembler Syntax: ROXL <ea>

Die 16 Bits des angewählten Speicherwortes werden um eine Position nach links rotiert.

Dazu gehört das folgende Format des Befehlswortes:

15	14	13	12	11	10	9	8	7	6	5 4 3	2 1 0
1	1	1	0	0	1	0	1	1	1	Effektive Adresse	
										Mode	Register

Bit 5..0 Wählt die Effektive Adresse des Quelloperanden an. Die folgenden Adressierungsarten sind erlaubt:

Adr.-Art	Mode	Reg
Dn	nicht erlaubt	
An	nicht erlaubt	
(An)	010	R:An
(An)+	011	R:An
-(An)	100	R:An
d16(An)	101	R:An
d8(An,Xi)	110	R:An

Adr.-Art	Mode	Reg
xxx.W	111	000
xxx.L	111	001
d16(PC)	nicht erlaubt	
d8(PC,Xi)	nicht erlaubt	
#<data>	nicht erlaubt	
Erläuterung siehe Kapitel 5		

Für die Adressierungsart Dn siehe auf der vorherigen Seite.

Benutzen Sie

ROXR: wenn nach rechts rotiert werden soll.
ROL: Wenn die Rotation nicht durch das X-Bit laufen soll.

Siehe auch: ASL , ASR, LSL, LSR, ROR, SWAP

Rotiere rechts mit Extend-Bit
rotate right with extend

ROXR

Ziel rotiert durch <Zahl> -> Ziel

Operandgröße: ROXR.B Byte (8 Bit)
ROXR.W Wort (16 Bit)
ROXR.L Langwort (32 Bit)
ROXR Wort (16 Bit)

Assembler Syntax:

```
ROXR.x  Dn, Dn         (Quelle, Ziel)
ROXR.x  #<data>, Dn
    (x entspricht B, W, L)
ROXR    <ea>
```

Beschreibung:

Die Bits des Operanden werden nach rechts rotiert. Bei jedem Rotationsschritt passiert folgendes:

- o Bit 0 gibt seinen Wert an das C-Bit ab. Der Wert wird außerdem zwischengespeichert.
- o Danach gibt Bit 1 seinen Wert an Bit 0, und Bit 2 seinen Wert an Bit 1 usw., bis das hochwertige Bit seinen Wert an seinen rechten Nachbarn abgibt.
- o Dann erhält das hochwertige Bit den Wert des X- Bit.
- o Zum Schluß erhält das hochwertige Bit den Wert, der vorher im C-Bit war.

Für die Gesamtzahl der Rotationsschritte siehe auf der nächsten Seite.

ROXR

Rotiere rechts mit Extend-Bit
rotate right with extend



Es gibt drei Befehlsformen.

o Der Befehl
ROXR.x #<data>, Dn (x entspricht B, W, L)
rotiert ein Datenregister nach rechts um soviele Positionen, wie in der Konstante angegeben ist. Die Maximalzahl ist acht.

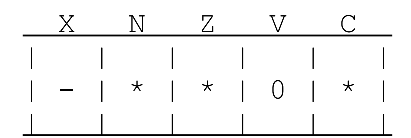
o Der Befehl
ROXR.x Dn, Dn (x entspricht B, W, L)
rotiert ein Datenregister nach rechts. Ein zweites Datenregister legt fest, um wieviele Positionen rotiert wird.

o Der Befehl
ROXR <ea>
rotiert eine Speicherstelle (16 Bit) um eine Position nach rechts.

Rotiere rechts mit Extend-Bit
rotate right with extend

ROXR

Condition Code Register:



X	N	Z	V	C
-	*	*	0	*

C erhält den Wert, der zuletzt aus dem hochwertigen Bit des Operanden herausgeschoben wurde. Wird zurückgesetzt beim Rotieren um Null Positionen.
V wird zurückgesetzt.
Z wird gesetzt, wenn das Ergebnis gleich Null ist. Wird sonst zurückgesetzt.
N wird gesetzt, wenn das hochwertige Bit des Ergebnisses gesetzt ist. Wird sonst zurückgesetzt.
X erhält den Wert, der zuletzt aus dem hochwertigen Bit des Operanden herausgeschoben wurde. Bleibt unverändert bei Schieben um Null Positionen.

ROXR

Rotiere rechts mit Extend-Bit
rotate right with extend

Assembler Syntax: ROXR.x Dn, Dn
ROXR.x #<data>, Dn
(x entspricht B, W, L)

Dazu gehört das folgende Format des Befehlswortes:

15	14	13	12	11 10 9	8	7 6	5	4	3	2 1 0
1	1	1	0	Zähl-Register	0	Größe	i	1	0	Daten-Register

Bit 11..9 Zählregister-Feld
Wenn i = 0:
Die Bits 11..9 geben an, um wieviele Positionen die Bits des Zieloperanden nach rechts rotiert werden. Dabei entspricht 001 einer Position usw., bis 111 sieben Positionen entspricht. 000 entspricht aber acht Positionen. (Konstante)
Wenn i = 1:
Die Bits 11..9 wählen ein Datenregister Dn an. Die niederwertigen 6 Bits des Datenregisters Dn geben an, um wieviele Positionen die Bits des Zieloperanden nach rechts rotiert werden.

Bit 7..6 Größe-Feld
00 Byte-Befehl ROXR.B
01 Wort-Befehl ROXR.W
10 Langwort-Befehl ROXR.W

Bit 5 i-Feld
0 Die Bits 11..9 beziehen sich auf eine Konstante
1 Die Bits 11..9 beziehen sich auf ein Datenregister.

Bit 2..0 Register-Feld
wählt ein Datenregister als Zieloperand an.

Rotiere rechts mit Extend-Bit
rotate right with extend

ROXR

Assembler Syntax: ROXR <ea>

Die 16 Bits des angewählten Speicherwortes werden um eine Position nach rechts rotiert.

Dazu gehört das folgende Format des Befehlswortes:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	0	0	1	1	Effektive Adresse					
										Mode			Register		

Bit 5..0 Wählt die Effektive Adresse des Quelloperanden an. Die folgenden Adressierungsarten sind erlaubt:

Adr.-Art	Mode	Reg
Dn	nicht erlaubt	
An	nicht erlaubt	
(An)	010	R:An
(An)+	011	R:An
-(An)	100	R:An
d16(An)	101	R:An
d8(An,Xi)	110	R:An

Adr.-Art	Mode	Reg
xxx.W	111	000
xxx.L	111	001
d16(PC)	nicht erlaubt	
d8(PC,Xi)	nicht erlaubt	
#<data>	nicht erlaubt	
Erläuterung siehe Kapitel 5		

Für die Adressierungsart Dn siehe auf der vorherigen Seite.

Benutzen Sie
ROXL: wenn nach links rotiert werden soll.
ROR: Wenn die Rotation nicht durch das X-Bit laufen soll.

Siehe auch: ASL, ASR, LSL, LSR, ROL, SWAP


RTE

Rückkehr von Exception
return from exception
(privilegierter Befehl)

Wenn Supervisor Mode:
(SP) -> SR; SP + 2 -> SP;
(SP) -> PC; SP + 4 -> SP

Wenn User Mode: **Auslösung Exception 8 (Kap 6)**
(Verletzung Privilegium)

Operandgröße: keine

Assembler Syntax: RTR

Beschreibung:
Das Status Register und der Programmzähler wird vom Stack gepopped. Das alte Status Register und der alte Programmzähler gehen verloren.

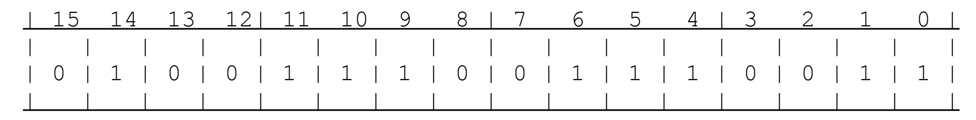
Mit diesem Befehl wird eine Exception beendet. Nach dem Befehl kann der Prozessor sich im User Mode befinden, abhängig davon, welcher Wert in Bit 13 des Status Registers gepopped wurde.

HINWEIS: Auf den Prozessoren 68010 und 68020 verhält sich RTE anders.

X	N	Z	V	C
*	*	*	*	*

Condition Code Register:
übernimmt den Wert vom Stack

Format des Befehlswortes:



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	0	0	1	1

($4E73)

Rückkehr Unterprogramm + Rückladen CCR
return and restore condition codes **RTR**

(SP) -> CCR; SP + 2 -> SP;
(SP) -> PC; SP + 4 -> SP

Operandgröße: keine

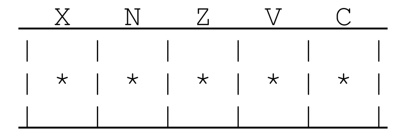
Assembler Syntax: RTR

Beschreibung:

Das Condition Code Register und der Programmzähler wird vom Stack gepopped. Das alte Condition Code Register und der alte Programmzähler gehen verloren. Das System Byte bleibt aber unverändert.

PROGRAMMIERHINWEIS:
Mit diesem Befehl beenden Sie ein Unterprogramm, das Sie z.B. so aufgerufen haben:

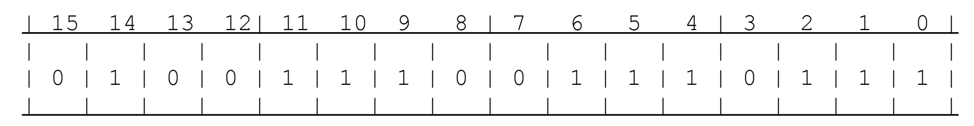
```
     MOVE.W SR, -(SP) ; nur für 68000, siehe Bemer-
                      ;   kung bei MOVE SR
     JSR SUBROUTINE   ; bzw. BSR, nach Ihrem Wunsch
```



X	N	Z	V	C
*	*	*	*	*

Condition Code Register:
übernimmt den Wert vom Stack

Format des Befehlswortes:



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	0	1	1	1

($4E77)

RTS

Rückkehr vom Unterprogramm
return from subroutine

(SP) -> PC; SP + 4 -> SP

Operandgröße: keine

Assembler Syntax: RTS

Beschreibung:

Der Programmzähler wird vom Stack gepopped. Der alte Programmzähler geht verloren.

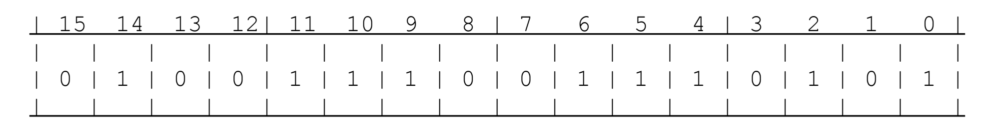
PROGRAMMIERHINWEIS:

Mit diesem Befehl beenden Sie ein Unterprogramm, das Sie mit BSR oder JSR aufgerufen haben.

X	N	Z	V	C
-	-	-	-	-

Condition Code Register:
bleibt unverändert

Format des Befehlswortes:



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	0	1	0	1

($4E75)

Subtrahiere BCD-Zahl mit Extend-Bit
subtract decimal with extend

SBCD

$Quelle_{10}$ - $Ziel_{10}$ - X -> Ziel

Operandgröße: SBCD.B Byte (8 Bit)
(ein Byte enthält zwei BCD-Zahlen)

Assembler Syntax:
SBCD.B Dn, Dn
SBCD.B -(An), -(An) (Quelle, Ziel)

Beschreibung:
Der Quelloperand und das X-Bit werden vom Zieloperand subtrahiert: das Ergebnis wird im Zieloperand abgespeichert. Die Subtrahierung findet als BCD-Arithmetik statt.

Die Operanden können in zwei Arten adressiert werden:

Dn Datenregister zu Datenregister. Die Operanden sind die niederwertigsten Bytes des Datenregisters.

-(An) Von Speicherplatz zu Speicherplatz. Diese Art ist gedacht, um mehrere BCD-Zahlen im Speicher zu subtrahieren. Die Operanden werden durch das Adressregister in Prä-dekrement-Mode adressiert.

Da der 68000 BCD-Zahlen mit dem niederwertigsten Byte auf der höchsten Speicherstelle ablegt, können Sie auf der höchsten Adresse anfangen, um mehrere Bytes automatisch abzuarbeiten.

Für eine weitere Beschreibung siehe bei NBCD.

Für eine Darstellung von BCD-Ziffern siehe Kap. 3.6

SBCD

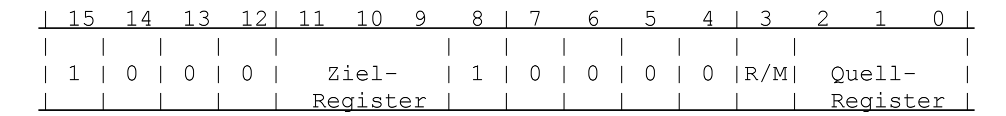
Subtrahiere BCD-Zahl mit Extend-Bit
subtract decimal with extend

X	N	Z	V	C
*	?	*	?	*

Condition Code Register:

C wird gesetzt wenn eine (dezimale) "Leihe" generiert wird. Wird sonst zurückgesetzt.
V nicht definiert
Z wird gelöscht, wenn das Ergebnis ungleich Null ist. Bleibt sonst unverändert.
N nicht definiert
X Erhält den gleichen Wert wie das C-Bit.

Format des Befehlswortes:



15	14	13	12	11 10 9	8	7	6	5	4	3	2 1 0
1	0	0	0	Ziel-Register	1	0	0	0	0	R/M	Quell-Register

Bit 11..9 Registerfeld: wählt eines der acht Daten- oder Adressregistern als Zieloperand an.

Bit 3 Das A-bit wählt die Adressierungsart an.
0 Adressierungsart Dn: Die Operation erfolgt von Datenregister zu Datenregister.
1 Adressierungsart -(An): Die Operation erfolgt von Speicherplatz zu Speicherplatz. Die Operanden werden durch das Adressregister in Prä-dekrement-mode adressiert. Siehe Kap. 5.

Bit 2..0 Registerfeld: wählt einer der acht Daten- oder Adressregistern als Quelloperand an.

Siehe auch: ABCD, NBCD, SUBX

Setze Byte aufgrund Bedingung
set according to condition

Scc

Scc ist ein überbegriff,
siehe Befehlsliste

```
Wenn (Bedingung = wahr),  dann  11111111 -> Ziel
                          sonst 00000000 -> Ziel
```

Operandgröße: Byte (8 Bit)

Assembler Syntax: Scc <ea>

Beschreibung:
Die Bezeichnung Scc im Kopf dieser Seite ist stellvertretend für die Befehle SCC, SCS, SEQ, SGE, SGT, SHI, SLE, SLS, SLT, SMIr SNE, SPL, SVC, SVS, SF und ST. Wir fassen alle diese Befehle hier zusammen.

Mit den Befehlen Scc können Sie Ergebnisse von Vergleichen WAHR oder UNWAHR (sog. Boolean-Variablen) im Speicher oder in ein Datenregister abspeichern. Es wird dann abgespeichert:

für WAHR (TRUE)
- 11111111 binär oder $FF hexadezimal
- Wir bezeichnen das als SETZEN eines Bytes.

für UNWAHR (FALSE)
- 00000000 binär oder $00 hexadezimal
- Wir bezeichnen das als ZURÜCKSETZEN eines Bytes.

Scc

Setze Byte aufgrund Bedingung
set according to condition

Bei den Befehlen werden die Bits N (Negative), Z (Zero), V (oVerflow) und C (Carry) des Condition Code Registers benutzt.

Die Bedingungen, die geprüft werden, sind ähnlich den Bedingungen für bedingte Sprünge - siehe bei Bcc.

HINWEISE:

Wenn Sie für UNWAHR (FALSE) die Zahl 100000000 binär bzw. $80 bevorzugen, erzeugen Sie das Ergebnis, indem Sie dem Befehl Scc <ea> noch den Befehl NEG <ea> nachschalten.

Sie können WAHR-UNWAHR-Information auch platzsparend in einem einzelnen Bit abspeichern, indem Sie jeweils einen der Befehle BSET oder BCLR benutzen. Sie brauchen dazu dann einen bedingten Sprung der Sorte Bcc, um den richtigen der beiden Befehle anzuspringen. Dazu müßten Sie dann auch das Zielbit anwählen. Dieser Aufwand lohnt sich aber erst bei einer größeren Menge von WAHR-UNWAHR-Daten.

Setze Byte aufgrund Bedingung
set according to condition

Scc

SCC Setze <ea>, wenn C-Bit zurückgesetzt ist
Set <ea> if Carry is Clear
Wenn die Bedingung nicht erfüllt ist, wird das Byte in <ea> zurückgesetzt.

SCS Setze <ea>, wenn C-Bit gesetzt ist
Set <ea> if Carry is Set
Wenn die Bedingung nicht erfüllt ist, wird das Byte in <ea> zurückgesetzt.

SEQ Setze <ea>, wenn gleich.
Set <ea> if EQual
Das Byte in <ea> wird gesetzt, wenn das Z-Bit (Zero) gesetzt ist. Sonst wird das Byte in <ea> zurückgesetzt.

SGE Setze <ea>, wenn größer oder gleich.
Set <ea> on Greater than or Equal
Das Byte in <ea> wird gesetzt, wenn das N-Bit (Negative) und das V-Bit (oVerflow) entweder beide gesetzt oder beide zurückgesetzt sind. Sonst wird das Byte in <ea> zurückgesetzt. SGE ist für Binärzahlen mit Vorzeichen gedacht.

Scc

Setze Byte aufgrund Bedingung
set according to condition

SGT Setze <ea>, wenn größer
Set <ea> on Greater Than

Das Byte in <ea> wird gesetzt, wenn

- o das M-Bit und das V-Bit gesetzt sind und das Z-Bit zurückgesetzt ist,
  oder
- o das N-Bit, das V-Bit und das Z-Bit alle zurückgesetzt sind.

Sonst wird das Byte in <ea> zurückgesetzt.
SGT ist für Binärzahlen mit Vorzeichen gedacht, ist sonst ähnlich SHI.

SHI Setze <ea>, wenn höher
Set <ea> on Higher than

Das Byte in <ea> wird gesetzt, wenn das C-Bit und das Z-Bit beide zurückgesetzt sind. Sonst wird das Byte in <ea> zurückgesetzt.
SGE ist für Binärzahlen ohne Vorzeichen gedacht, ist sonst ähnlich SGT.

SLE Setze <ea>, wenn kleiner oder gleich
Set <ea> on Less than or Equal

Das Byte in <ea> wird gesetzt, wenn

- o das Z-Bit gesetzt ist,
  oder
- o das N-Bit gesetzt und das V-Bit zurückgesetzt ist,
  oder
- o das N-Bit zurückgesetzt und das V-Bit gesetzt ist.

Sonst wird das Byte in <ea> zurückgesetzt.
SLE ist für Binärzahlen mit Vorzeichen gedacht, ist sonst ähnlich SLS.

Setze Byte aufgrund Bedingung
set according to condition

Scc

SLS Setze <ea>, wenn niedriger oder gleich
Set <ea> on Lower or Same
Das Byte in <ea> wird gesetzt, wenn das C-Bit, das Z-Bit oder beide gesetzt sind. Sonst wird das Byte in <ea> zurückgesetzt.
SLS ist für Binärzahlen ohne Vorzeichen gedacht, ist sonst ähnlich SLE.

SLT Setze <ea>, wenn kleiner
Set <ea> on Less Than
Das Byte in <ea> wird gesetzt, wenn
- o das N-Bit gesetzt und das V-Bit zurückgesetzt,

oder
- o das N-Bit zurückgesetzt und das V-Bit gesetzt ist.

Sonst wird das Byte in <ea> zurückgesetzt.
SLT ist für Binärzahlen mit Vorzeichen gedacht.

SMI Setze <ea>, wenn Minus
Set <ea> on Minus
Das Byte in <ea> wird gesetzt, wenn das N-Bit gesetzt ist.
Sonst wird das Byte in <ea> zurückgesetzt.
SMI ist für Binärzahlen mit Vorzeichen gedacht.

SNE Setze <ea>, wenn ungleich
Set <ea> on Not Egual
wenn das Z-Bit
zurückgesetzt ist.



Setze Byte aufgrund Bedingung
set according to condition

SPL **Setze <ea> r wenn Plus**
Set <ea> on PLus
Das Byte in <ea> wird gesetzt, wenn das N-Bit zurückgesetzt ist.
Sonst wird das Byte in <ea> zurückgesetzt.
SPL ist für Binärzahlen mit Vorzeichen gedacht.

SVC **Setze <ea>, wenn kein Überlauf**
Set <ea> on oVerflow Clear
Das Byte in <ea> wird gesetzt, wenn das V-Bit zurückgesetzt ist. Sonst wird das Byte in <ea> zurückgesetzt.

SVS **Setze <ea>, wenn Überlauf**
Set <ea> on oVerflow Set
Das Byte in <ea> wird gesetzt, wenn das V-Bit gesetzt ist. Sonst wird das Byte in <ea> zurückgesetzt.

SF **Setze <ea> nie**
never set <ea>
Das Byte in <ea> wird zurückgesetzt, unabhängig von irgendwelcher Bedingung.

ST Setze <ea> immer
always set <ea>
Das Byte in <ea> wird gesetzt, unabhängig von irgendwelcher Bedingung.

X	N	Z	V	C
-	-	-	-	-

Condition Code Register:
bleibt unverändert

Setze Byte aufgrund Bedingung
set according to condition

Scc

Format des Befehlswortes:

15	14	13	12	11 10 9 8	7	6	5 4	3 2 1 0
0	1	0	1	Bedingung	1	1	Effektive Adresse Mode	Effektive Adresse Register

Bit 11..8 Bedingungsfeld

Bedingung	Befehl
0000	ST
0001	SF
0010	SHI
0011	SLS
0100	SCC
0101	SCS
0111	SEQ
0110	SNE

Bedingung	Befehl
1000	SVC
1001	SVS
1010	SPL
1011	SMI
1100	SGE
1101	SLT
1110	SGT
1111	SLE

Bit 5..0 Die Effektive Adresse wählt den Zieloperand an. Die folgenden Adressierungsarten sind erlaubt:

Adr.-Art	Mode	Reg
Dn	000	R:Dn
An	nicht erlaubt	
(An)	010	R:An
(An)+	011	R:An
-(An)	100	R:An
d16(An)	101	R:An
d8(An,Xi)	110	R:An

Adr.-Art	Mode	Reg
xxx.W	111	000
xxx.L	111	001
d16(PC)	nicht erlaubt	
d8(PC,Xi)	nicht erlaubt	
#<data>	nicht erlaubt	
Erläuterung siehe Kapitel 5		

STOP

Lade Status Register und Halt
load Status register and stop
(privilegierter Befehl)

Wenn Supervisor Mode: Konstante -> Status Register
Halt

Wenn User Mode: Auslösung Exception 8 (Kap 6)
(Verletzung Privilegium)

Operandgröße: keine

Assembler Syntax: STOP #<data>

Beschreibung:
Die Konstante, die dem Befehlswort unmittelbar folgt, wird im Status Register geladen. Danach findet keine Aktion mehr statt: der nächste Befehl wird nicht ausgeführt.

Die folgenden Ereignisse setzen den Prozessor wieder in Gang:

- Wenn mit dem Befehl STOP das T-Bit (Bit 15 des Status Registers) gesetzt wird, findet nach dem Befehl eine Trace-Exception (Exception 9) statt.

- Ein Interrupt. Voraussetzung ist natürlich, daß die Interruptebene mindestens den Wert der soeben gesetzten Interruptmaske I2..I0 hat.

- Eine Reset-Exception setzt den Prozessor zu jeder Zeit wieder im Gang.

Mit diesem Befehl können schwer zu determinierende Programmfehler gefunden werden.

Lade Status Register und Halt
load Status register and stop
(privilegierter Befehl)

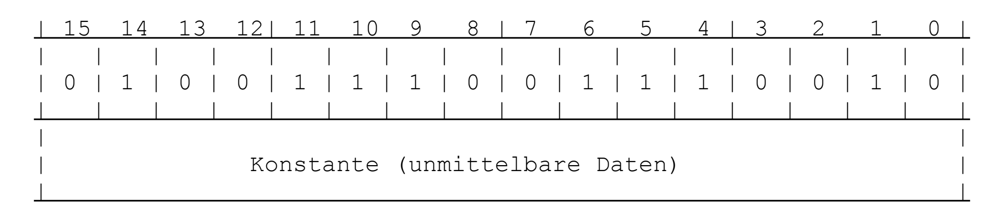


STOP

X	N	Z	V	C
*	*	*	*	*

Condition Code Register:
wird entsprechend gesetzt

Format des Befehlswortes:



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	0	0	1	0
Konstante (unmittelbare Daten)															

($4E72)

Siehe auch: TRAP, ILLEGAL, CHK



Subtrahiere binär
subtract binary

Ziel - Quelle -> Ziel

Operandgröße: SUB.B Byte (8 Bit)
SUB.W Wort (16 Bit)
SUB.L Langwort (32 Bit)

Assembler Syntax:	**Operation:**
SUB.x Dn, <ea>	Dn - <ea> -> Dn
SUB.x <ea>,Dn	<ea> - Dn -> <ea>
(x entspricht B, W, L)	

Beschreibung:
Der Quelloperand wird binär vom Zieloperand subtrahiert, und das Ergebnis wird im Zieloperand abgespeichert.

Einer der beiden Operanden muß ein Datenregister sein.

Die Größe des Operanden sowie die Angabe, welcher Operand das Datenregister ist, sind im Mode-Feld enthalten.

Subtrahiere binär
subtract binary

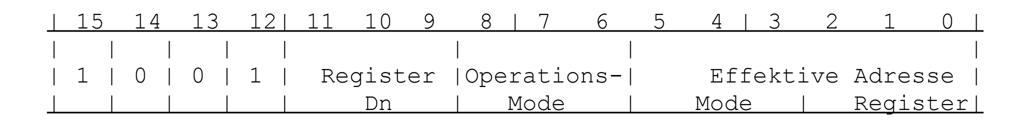
SUB

X	N	Z	V	C
*	*	*	*	*

Condition Code Register:

C wird gesetzt, wenn ein "Leihen" generiert wird.
Wird sonst zurückgesetzt.
V wird gesetzt, wenn ein Überlauf generiert wird.
Wird sonst zurückgesetzt.
Z wird gesetzt, wenn das Ergebnis gleich Null ist.
Wird sonst zurückgesetzt.
N wird gesetzt, wenn das Ergebnis negativ ist.
Wird sonst zurückgesetzt.
X Erhält den gleichen Wert wie das C-Bit.

Format des Befehlswortes:



15	14	13	12	11 10 9	8 7 6	5 4 3	2 1 0
1	0	0	1	Register Dn	Operations-Mode	Effektive Adresse Mode	Register

Bit 11..9 Registerfeld: wählt eines der acht Daten-register an.

Bit 8..6 Feld Operationsmode:

SUB.B	SUB.W	SUB.L	Operation
000	001	010	Dn - <ea> -> Dn
100	101	110	<ea> - Dn -> <ea>

SUB

Subtrahiere binär
subtract binary

Bit 5..0 Wenn die Effektive Adresse der Quelloperand ist (also Dn - <ea> -> Dn), sind die folgenden Adressierungsarten erlaubt:

Adr.-Art	Mode	Reg
Dn	000	R:Dn
An *)	001	R:An
(An)	010	R:An
(An)+	011	R:An
-(An)	100	R:An
d16(An)	101	R:An
d8(An,Xi)	110	R:An

Adr.-Art	Mode	Reg
xxx.W	111	000
xxx.L	111	001
d16(PC)	111	010
d8(PC,Xi)	111	011
#<data>	111	100
Erläuterung siehe Kapitel 5		

*) Adressierungsart An nicht für Byte-Befehle erlaubt

Bit 5..0 Wenn die Effektive Adresse der Zieloperand ist (also <ea> - Dn -> <ea>), sind die folgenden Adressierungsarten erlaubt:

Adr.-Art	Mode	Reg
Dn	nicht erlaubt	
An	nicht erlaubt	
(An)	010	R:An
(An)+	011	R:An
-(An)	100	R:An
d16(An)	101	R:An
d8(An,Xi)	110	R:An

Adr.-Art	Mode	Reg
xxx.W	111	000
xxx.L	111	001
d16(PC)	nicht erlaubt	
d8(PC,Xi)	nicht erlaubt	
#<data>	nicht erlaubt	
Erläuterung siehe Kapitel 5		

HINWEIS:
Wenn der Zieloperand ein Datenregister sein soll, kann er nicht mit der Adressierungsart <ea> angewählt werden, sondern nur mit der Adressierungsart Dn.

Subtrahiere binär
subtract binary

SUB

PROGRAMMIERHINWEIS:

Es kommt manchmal vor, daß Sie z.B. SUB.W Dl, D2 brauchen, aber trotzdem Dl behalten möchten. Programmieren Sie dann:

```
SUB.W  D2, Dl
NEG.W  Dl
```

Benutzen Sie

SUBA: wenn der Zieloperand ein Adress-register ist;

SUBI oder SUBQ: wenn einer der Operanden eine Konstante ist.

NEX oder NEGX: wenn Sie von Null subtrahieren möchten.

SUBX: wenn ein anderes Verhalten des Z-Bits erwünscht ist.

Siehe auch: ADD

SUBA

Subtrahiere Adresse
subtract address

Ziel - Quelle -> Ziel

Operandgröße: SUBA.W Wort (16 Bit)
SUBA.L Langwort (32 Bit)

Assembler Syntax: SUBA.x <ea>, An (Quelle, Ziel)
(x entspricht W, L)

Beschreibung:
Der Quelloperand in <ea> wird binär vom Ziel-Adressregister An subtrahiert.

Das Ergebnis wird im Ziel-Adressregister An abgespeichert.

Wenn die Operandgröße des Quelloperanden ein Wort ist, wird der Quelloperand mit dem gleichen Vorzeichen auf 32 Bit erweitert. Vom Zielregister An werden sämtliche Bytes angewendet, unabhängig von der Operandgröße.

X	N	Z	V	C
-	-	-	-	-

Condition Code Register:
keine Änderungen

Subtrahiere Adresse
subtract address

SUBA

Format des Befehlswortes:

15	14	13	12	11 10 9	8 7 6	5 4 3	2 1 0
1	0	0	1	Register An	Operations-Mode	Effektive Adresse Mode	Effektive Adresse Register

Bit 11..9 Registerfeld: wählt eines der acht Adressregister An an.
Es ist der Zieloperand.

Bit 8..6 Feld Operations-Mode:

011 SUBA.W - Wort-Befehl. Der Quellopperand wird mit dem gleichen Vorzeichen auf 32 Bit erweitert, und vom Ziel-Adressregister werden sämtliche 32 Bits angewendet.

111 SUBA.L - Langwort-Befehl

Bit 5..0 Die Effektive Adresse wählt den Quell-operand an. Alle Adressierungsarten sind erlaubt:

Adr.-Art	Mode	Reg
Dn	000	R:Dn
An	001	R:An
(An)	010	R:An
(An)+	011	R:An
-(An)	100	R:An
d16(An)	101	R:An
d8(An,Xi)	110	R:An

Adr.-Art	Mode	Reg
xxx.W	111	000
xxx.L	111	001
d16(PC)	111	010
d8(PC,Xi)	111	011
#<data>	111	100
Erläuterung siehe Kapitel 5		

Siehe auch: ADDA, SBCD, SUBA, SUBI, SUBQ, SUBX

SUBI

Subtrahiere Konstante
subtract immediate

Ziel - Konstante -> Ziel

Operandgröße: SUBI.B Byte (8 Bit)
SUBI.W Wort (16 Bit)
SUBI.L Langwort (32 Bit)

Assembler Syntax: SUBI.x #<data>, <ea> (Quelle, Ziel)
(x entspricht B, W, L)

Beschreibung:

Die Konstante, die im Speicher unmittelbar dem Befehlswort folgt, wird binär vom Zieloperand <ea> subtrahiert.

Das Ergebnis wird im Zieloperand <ea> abgespeichert.

Die Größe der Konstante entspricht der Operandgröße.

Condition Code Register:

X	N	Z	V	C
*	*	*	*	*

C wird gesetzt, wenn ein "Leihen" generiert wird. Wird sonst zurückgesetzt.
V wird gesetzt, wenn ein Überlauf generiert wird. Wird sonst zurückgesetzt.
Z wird gesetzt, wenn das Ergebnis gleich Null ist. Wird sonst zurückgesetzt.
N wird gesetzt, wenn das Ergebnis negativ ist. Wird sonst zurückgesetzt.
X Erhält den gleichen Wert wie das C-Bit.

Subtrahiere Konstante
subtract immediate

SUBI

Format des Befehlswortes:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	Größe		Effektive Adresse Mode			Register		
1. Argumentwort: Wort Daten								bzw. Byte Daten							
2. Argumentwort: Langwort-Daten (einschließlich voriges Wort)															

Bit 7..6 Größe-Feld:
00 Byte-Befehl SUBI.B
01 Wort-Befehl SUBI.W
10 Langwort-Befehl SUBI.L

Aufbau der Argumentwörter siehe Kap. 3.8

Bit 5..0 Die Effektive Adresse wählt den Zieloperand an. Die folgenden Adressierungsarten sind erlaubt:

Adr.-Art	Mode	Reg
Dn	000	R:Dn
An	nicht erlaubt	
(An)	010	R:An
(An)+	011	R:An
-(An)	100	R:An
d16(An)	101	R:An
d8(An,Xi)	110	R:An

Adr.-Art	Mode	Reg
xxx. W	111	000
xxx.L	111	001
d16(PC)	nicht erlaubt	
d8(PC,Xi)	nicht erlaubt	
#<data>	nicht erlaubt	
Erläuterung siehe Kapitel 5		

Argumentwort:
Bit 7..6 = 00 -> Datenfeld ist die niederwertige Hälfte des 1. Argumentwortes
Bit 7..6 = 01 -> Datenfeld ist das 1. Argumentwort
Bit 7..6 = 10 -> Datenfeld ist 1. +2. Argumentwort

Siehe auch: ADDI, SBCD, SUBX, SUBA, SUBQ, SUBX

SUBQ

Subtrahiere Konstante "quick" (1..8)
subtract quick

Ziel - Konstante -> Ziel

Operandgröße: SUBQ.B Byte (8 Bit)
SUBQ.W Wort (16 Bit)
SUBQ.L Langwort (32 Bit)

Assembler Syntax: SUBQ.x #<data>, <ea> (Quelle, Ziel)
(x entspricht B, W, L)

Beschreibung:

Dieser Befehl subtrahiert die Konstante vom Zieloperand <ea>. Das Ergebnis wird im Zieloperand <ea> abgespeichert.
Die Konstante muß zwischen 1 und 8 liegen.

Condition Code Register:

X	N	Z	V	C
*	*	*	*	*

C wird gesetzt, wenn ein "Leihen" generiert wird. Wird sonst zurückgesetzt.
V wird gesetzt, wenn ein Überlauf generiert wird. Wird sonst zurückgesetzt.
Z wird gesetzt, wenn das Ergebnis gleich Null ist. Wird sonst zurückgesetzt.
N wird gesetzt, wenn das Ergebnis negativ ist. Wird sonst zurückgesetzt.
X Erhält den gleichen Wert wie das C-Bit.

HINWEIS: Wenn der Zieloperand ein Adressregister ist (Adressierungsart An), wird das Condition Code Register nicht geändert.

Subtrahiere Konstante "quick" (1..8)
subtract quick

SUBQ

Format des Befehlswortes:

15	14	13	12	11 10 9	8	7 6	5 4 3	2 1 0
0	1	0	1	D a t e n	0	Größe	Effektive Adresse Mode	Register

Bit 11..9 Datenfeld:
000 entspricht Konstante 1, 001 entspricht Konstante 2 usw. bis 111 entspricht Konstante

Bit 7..6 Größe-Feld:
00 Byte-Befehl SUBQ.B
01 Wort-Befehl SUBQ.W
10 Langwort-Befehl SUBQ.L

Bit 5..0 Die Effektive Adresse wählt den Zieloperand an. Die folgenden Adressierungsarten sind erlaubt:

Adr.-Art	Mode	Reg
Dn	000	R:Dn
An *)	001	R:An
(An)	010	R:An
(An)+	011	R:An
-(An)	100	R:An
d16(An)	101	R:An
d8(An,Xi)	110	R:An

Adr.-Art	Mode	Reg
xxx.W	111	000
xxx.L	111	001
d16(PC)	nicht erlaubt	
d8(PC,Xi)	nicht erlaubt	
#<data>	nicht erlaubt	
Erläuterung siehe Kapitel 5		

*)Adressierungsart An nicht für Byte-Befehle erlaubt

Benutzen Sie
SUBI: wenn die Konstante außerhalb des Bereiches 1..8 liegt.

Siehe auch: SBCD, SUB, SUBA, SUBX, ADDQ, NEG



Subtrahiere mit Extend-Bit
subtract with extend

Ziel - Quelle - X -> Ziel

```
Operandgröße:  SUBX.B  Byte (8 Bit)
               SUBX.W  Wort (16 Bit)
               SUBX.L  Langwort (32 Bit)

Assembler      SUBX.x  Dn, Dn          (Quelle, Ziel)
Syntax:        SUBX.x -(An), -(An)
                   (x entspricht B,W,L)
```

Beschreibung:
Der Quelloperand und das Extend-Bit werden vom Zieloperand subtrahiert. Das Ergebnis wird im Zieloperand abgespeichert.

Die Operanden können in zwei Arten adressiert werden:

Dn Datenregister zu Datenregister. Die Operanden sind die niederwertigsten Bytes der Datenregister.

-(An) Von Speicherplatz zu Speicherplatz. Diese Art ist gedacht, um mehrere Binärzahlen im Speicher zu subtrahieren. Die Operanden werden durch das Adressregister in Prä-dekrement-Mode adressiert.

Da der 68000 Daten mit dem niederwertigsten Byte auf der höchsten Speicherstelle ablegt, können Sie auf der höchsten Adresse anfangen, um mehrere Bytes automatisch abzuarbeiten.

Für eine weitere Beschreibung siehe bei NEGX.

Subtrahiere mit Extend-Bit
subtract with extend

SUBx

Condition Code Register:

X	N	Z	V	C
*	*	*	*	*

C wird gesetzt, wenn ein Übertrag generiert wird. Wird sonst zurückgesetzt.
V wird gesetzt, wenn ein Überlauf generiert wird. Wird sonst zurückgesetzt.
Z wird gelöscht, wenn das Ergebnis ungleich Null ist. Bleibt sonst unverändert.
N wird gesetzt, wenn das Ergebnis negativ ist. Wird sonst zurückgesetzt.
X Erhält den gleichen Wert wie das C-Bit.

SUBx

Subtrahiere mit Extend-Bit
subtract with extend

Format des Befehlswortes:

15	14	13	12	11 10 9	8	7	6	5	4	3	2 1 0
1	0	0	1	Ziel-Register	1	0	0	0	0	A	Quell-Register

Bit 11..9 Registerfeld: wählt eines der acht Daten- oder Adressregister als Zieloperand an.

Bit 3 Das A-Bit wählt die Adressierungsart an.

0 Adressierungsart Dn: Die Operation erfolgt von Datenregister zu Datenregister

1 Adressierungsart -(An): Die Operation erfolgt von Speicherplatz zu Speicherplatz. Die Operanden werden durch das Adressregister in Prädekrement-mode adressiert. Siehe Kap. 5.

Bit 2..0 Registerfeld: wählt eines der acht Daten- oder Adressregister als Quelloperand an.

Siehe auch: ADDX, SUBI , SBCD, SUB, SUBA, SUBQ

Vertausche Register-Hälften
swap register halves

SWAP

Register[16..31] <-> Register[0..15]

Operandgröße: Wort (16 Bit)

Assembler Syntax: SWAP Dn

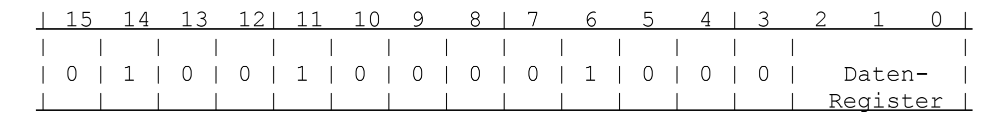
Beschreibung:
Die Hälften eines Datenregisters werden vertauscht.

X	N	Z	V	C
-	*	*	0	0

Condition Code Register:

C wird zurückgesetzt.
V wird zurückgesetzt.
Z wird gesetzt, wenn alle Bits des Ergebnisses Null sind. Wird sonst zurückgesetzt.
N wird gesetzt, wenn Bit 31 des Ergebnisses gesetzt ist. Wird sonst zurückgesetzt.
X ändert sich nicht.

Format des Befehlswortes:



15	14	13	12	11	10	9	8	7	6	5	4	3	2 1 0
0	1	0	0	1	0	0	0	0	1	0	0	0	Daten-Register

Bit 2..0 Register-Feld
wählt ein Datenregister als Operand an.

Siehe auch:
EXG, LSL, LSR, MOVE, ROL, ROR

TAS

Prüfe und setze Operand
test and set an Operand

Ziel wird geprüft -> Condition Code Register
l -> Bit 7 des Zieloperanden

Operandgröße: Byte (8 Bit)

Assembler Syntax: TAS <ea>

Beschreibung:
Der Zieloperand wird geprüft und das Condition Code wird entsprechend gesetzt.

Danach wird Bit 7 des Zieloperanden gesetzt.

Der Befehl TAS ist unteilbar. In einem System mit mehreren Prozessoren kann TAS auch benutzt werden, um die Prozessoren zu synchronisieren.

Condition Code Register:

X	N	Z	V	C
-	*	*	0	0

C wird zurückgesetzt.
V wird zurückgesetzt.
Z wird gesetzt, wenn der Operand Null war. Wird sonst zurückgesetzt.
N wird gesetzt, wenn der Operand negativ war (Bit 7 gesetzt). Wird sonst zurückgesetzt.
X ändert sich nicht.

Prüfe und setze Operand
test and set an Operand

TAS

Format des Befehlswortes:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	1	0	1	1	Effektive Adresse					
										Mode			Register		

Bit 5..0 wählt die effektive Adresse des Operanden an. Die folgenden Adressierungsarten sind erlaubt:

Adr.-Art	Mode	Reg
Dn	000	R:Dn
An	nicht erlaubt	
(An)	010	R:An
(An)+	011	R:An
-(An)	100	R:An
d16(An)	101	R:An
d8(An,Xi)	110	R:An

Adr.-Art	Mode	Reg
xxx.W	111	000
xxx.L	111	001
d16(PC)	nicht erlaubt	
d8(PC,Xi)	nicht erlaubt	
#<data>	nicht erlaubt	
Erläuterung siehe Kapitel 5		

Benutzen Sie

BSET: Wenn Sie ein anderes Bit (nicht nur Bit 7) des Operanden prüfen und setzen möchten.

BCLR: Wenn Sie ein Bit prüfen und zurücksetzen möchten.

TRAP

Trap
trap

Es wird eine der Exceptions 32..47 ausgelöst.

Operandgröße: keine

Assembler Syntax: TRAP #<Vektor>

Beschreibung:
Es wird eine der Exceptions 32..47 ausgelöst. Welche Exception ausgelöst wird, richtet sich nach der Vektornummer des TRAP-Befehls. Siehe Kap. 6.

TRAP-Vektor	Exception Vektor	Exception Adresse
0	32	$0080
1	33	$0084
2	34	$0088
3	35	$008C
4	36	$0090
5	37	$0094
6	38	$0098
7	39	$009C

TRAP-Vektor	Exception Vektor	Exception Adresse
8	40	$00A0
9	41	$00A4
10	42	$00A8
11	43	$00AC
12	44	$00B0
13	45	$00B4
14	46	$00B8
15	47	$00BC

X	N	Z	V	C
-	-	-	-	-

Condition Code Register:
bleibt unverändert

Trap
trap

TRAP

Format des Befehlswortes:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	0	0	Vektor			

($4E4x)

Feld Vektor:
legt fest, welche Exception (32..47) ausgelöst wird.

TRAF eignet sich dazu, um Befehle des Betriebssystems zu implementieren.

Eine ähnliche Wirkung haben die Axxx- und Fxxx-Emulationen, siehe bei Exception 10 und 11 nach.

Siehe auch: TRAPV, ILLEGAL

TRAPV

Trap, wenn Überlauf
trap on overflow

Wenn V-Bit gesetzt, dann wird Exception 7 ausgelöst

Operandgröße: keine

Assembler Syntax: TRAPV

Beschreibung:

Wenn das V-Bit (Überlauf) gesetzt ist, wird die Exception 7 ausgelöst (siehe Kap. 6).

Wenn das V-Bit (Überlauf) dagegen zurückgesetzt ist, findet keine Aktion statt. Der nächste Befehl im Speicher wird dann ausgeführt.

X	N	Z	V	C
-	-	-	-	-

Condition Code Register:
bleibt unverändert

Format des Befehlswortes:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	1	1	0	0

($4E76)

Benutzen Sie
CHK: wenn eine Grenzüberschreitung eines Datenregisters eine Exception verursachen soll.

Siehe auch: TRAP, ILLEGAL

Prüfe Operand
test an Operand

TST

Prüfe Operand -> Condition Code Register

Operandgröße: TST.B Byte (8 Bit)
TST.W Wort (16 Bit)
TST.L Langwort (32 Bit)

Assembler Syntax: TST.x <ea>
(x entspricht B, W, L)

Beschreibung:

Der Operand wird mit Null verglichen und das Condition Code Register wird entsprechend gesetzt. Der Operand wird nicht verändert.

Condition Code Register:

X	N	Z	V	C
-	*	*	0	0

C wird zurückgesetzt.
V wird zurückgesetzt.
Z wird gesetzt, wenn das Ergebnis gleich Null ist.
Wird sonst zurückgesetzt.
N wird gesetzt, wenn das hochwertige Bit des Ergebnisses gesetzt ist (zeigt ein negatives Ergebnis).
Wird sonst zurückgesetzt.
X bleibt unverändert.

TST

Prüfe Operand
test an Operand

Format des Befehlswortes:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	1	0	Größe		Effektive Adresse					
										Mode			Register		

Bit 7..6 Größe-Feld:
00 Byte-Befehl TST.B
01 Wort-Befehl TST.W
10 Langwort-Befehl TST.L

Aufbau der Argumentwörter siehe Kap. 3.8

Bit 5..0 Die Effektive Adresse wählt den Zieloperand an. Die folgenden Adressierungsarten sind erlaubt:

Adr.-Art	Mode	Reg
Dn	000	R:Dn
An	nicht erlaubt	
(An)	010	R:An
(An)+	011	R:An
-(An)	100	R:An
d16(An)	101	R:An
d8(An,Xi)	110	R:An

Adr.-Art	Mode	Reg
xxx.W	111	000
xxx.L	111	001
d16(PC)	nicht erlaubt	
d8(PC,Xi)	nicht erlaubt	
#<data>	nicht erlaubt	
Erläuterung siehe Kapitel 5		

Benutzen Sie
TSTI:
wenn einer der Operanden eine Konstante ist.

Siehe auch: OR, EOR, NOT, TST

Löse Reservierung im Stack auf
unlink



An -> SP; (SP) -> An; SP + 2 -> SP

Operandgröße: keine
Assembler Syntax: UNLK An

Beschreibung:

Der Stack Pointer erhält den Wert des angegebenen Adressregisters. Danach erhält das Adressregisters An den Wert, der aus dem Stack gepopped wird.

Der Befehl UNLK macht genau das entgegengesetzte als der Befehl LINK. Der von LINK reservierte Stackbereich (das Stack Frame) wird wieder freigegeben. Auch erhält das Adressregister An seinen früheren Wert wieder.

X	N	Z	V	C
-	-	-	-	-

Condition Code Register:
bleibt unverändert

Format des Befehlswortes:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	0	1	1	Adress-Register		

Bit 2..0 Feld Adressregister
spezifiziert, welches Adressregister für den Auflösungsbefehl benutzt werden soll.

B. Befehlsübersicht

Anhang C

Bedienungsanleitung der
mitgelieferten Programmdiskette

Im Public-Domain-Bereich haben wir eine Diskette gefunden mit einer interaktiven Debugger-Software. Diese Diskette legen wir dem Buch bei, damit der Leser das Erlernte schnell auf seinem Atari ausprobieren kann. Selbstverständlich wird weder vom Verlag, noch vom Autor dieses Buches irgendwelche Gewährleistung für die Software übernommen.

Für Ihren Comfort haben wir die sich auf der Diskette befindliche Dokumentationsdatei hier aufgenommen.

Beschreibung des Assemblers

1. Allgemeines

1.1 Dieser Assembler erlaubt das Erstellen von Maschinensprach-programmem unter Verwendung der allgemein gebräuchlichen 68000er Befehle. Er arbeitet zeilenorientiert, kann aber auch Dateien, die mit einem anderen Editor erstellt wurden, verarbeiten. Bei der Programmierung wurde auf einfache Handhabung Wert gelegt. Das Quellisting bleibt stets im Speicher, ebenso wird auch das beim Assemblieren entstandene Programm in den Speicher geschrieben. Von dort läßt es sich sofort aufrufen und testen.

1.2 Der Assembler wurde für die mittlere und hohe Auflösung geschrieben. Beim Gebrauch der niedrigen könnten (insbesondere beim Editieren) unerwünschte Effekte auftreten.

1.3 Das im Speicher befindliche Listing darf aus max. 30000 Zeilen bestehen. Hinzu kommt, daß Sie genug Speicher für das Listing, die Labels und das assemblierte Programm reserviert haben müssen. Um dies zu tun, wird gleich nach dem Laden der freie Speicher angezeigt. Sie haben nun die Möglichkeit, den Speicher nach Ihren Wünschen auf die drei Bereiche aufzuteilen. In der Regel brauchen Sie sich keine Gedanken zu machen und können durch Druck auf "Return" die Standartwerte einstellen.
Bedenken Sie jedoch, daß, falls der reservierte Speicher nicht ausreichen sollte, keine Meldung erfolgt und daß es zu Datenverlust kommen kann.

1.4. Der Assembler unterscheidet zwischen Groß- und Kleinbuchstaben nur dann, wenn diese in Anführungszeichen auftauchen. Dies ist einerseits der Fall, wenn Sie mit DC Strings definieren, oder wenn Sie mit LIST "LABEL" ab einem Label listen wollen.

2. Direktbefehle
Nach dem Laden des Assemblers und der Eingabe des zu reservierenden Speichers befinden Sie sich im Direktmodus. Es steht Ihnen nun eine Reihe von Befehlen zur Verfügung, die hier - nach ihren Funktionen sortiert - aufgeführt sind:

2.1 Befehle zum Bearbeiten des Programmlistings

ADD
Hiermit können Sie Zeilen an das Ende des Programms anfügen, bzw.,falls kein Programm im Speicher steht, ein neues eingeben.

E oder EDIT (Ln#, Ln#)

Dieser Befehl dient zum Ändern von Zeilen. Sie können die Start- und Endzeile angeben (durch ein Komma getrennt), auf eine oder beide Zeilenangabe(n) verzichten, usw. Statt einer Zeile können Sie auch in Anführungszeichen eine Zeichenfolge (z.B. ein Label) angeben. Das Editieren beginnt dann in der Zeile, an deren Anfang diese Zeichen stehen.

INS (Ln#)
Diese Funktion erlaubt es Ihnen, Zeilen vor der angegebenen Zeile einzufügen.

DEL (Ln#, Ln#)
Hiermit können Sie Bereiche des Programms löschen. Die Zeilenangaben funktionieren wie bei EDIT.

COPY Ln# TO Ln#
Dieser Befehl kopiert den angegebenen Bereich unmittelbar vor die Zeile, die hinter TO steht.

L oder LIST (Ln#, Ln#) Programm listen.

FIND "String" (Ln#, Ln#)
Diese Funktion sucht einen String im Listing.

NEW
Programm löschen

2.2 Assemblieren und Starten des Programmes

ASM (F)
Dieser Befehl assembliert. das Listing so, daß ein ablauffähiges Programm im Speicher erzeugt wird. Wenn Sie hinter ASM ein F schreiben, wird das Listing beim Assemblieren nicht angezeigt.

PRGASM (F)
Hiermit wird das Listing so assembliert, daß man das entstandene Programm anschließend mit PRGSAVE als Programmdatei speichern kann.

LABEL (Adr./Label)
Dieser Befehl zeigt alle Labels nach ihren Adressen geordnet an. Wenn Sie ein Label bzw. eine Adresse angeben, so werden die vorausgehenden Labels nicht gezeigt.

GO Adr./Label
Hiermit können Sie Ihr Programm aufrufen. Dabei müssen Sie die Startadresse (als Zahl oder Label) angeben.

TRACE Adr./Label
Aufgerufen wird dieser Befehl wie GO. Er zeigt allerdings nach jedem Maschinensprachbefehl die Registerinhalte und Flags an.

2.3. Befehle zum Speichern und Laden

DIR (Filespec.)
Directory zeigen. Sie können dabei beliebig mit Wild Cards arbeiten. Sollten Sie keine Angaben machen, wird *.* angenommen.

PATH (Drive:)Pathname
Hiermit können Sie das Laufwerk und den Pfadnamen festlegen.
(Beispiel: PATH Arpath.nam).

SAVE Filename Speichert das Programm

LOAD Filename Lädt ein Programm

MERGE Filename
Hängt ein Programm an das Ende des im Speicher stehende an.

BSAVE Adr., Length, Filename Sichert einen Bereich des Speichers.

BLOAD Adr,. Filename
Lädt eine Datei ab einer bestimmten Adresse in den Speicher.

PRGSAVE Filename
Dieser Funktion muß ein PRGASM vorausgehen. Sie speichert das assemblierte Progamm als ablauffähige, vom Desktop aus ladbare Datei

2.4 Druckeransteuerung

PON
Schaltet die Druckerausgabe ein: Alle Ausgaben auf dem Bildschirm werden gleichzitig an den Drucker gesandt.

POFF
Schaltet die Druckerausgabe ab.

2.5 Sonstige Befehle

DUMP Adr./ Label
Hiermit lassen sich Speicherinhalte ausgeben.

DIS Adr./label
Disassembliert den Speicher ab der agegebenen Adresse.

DEZHEX Expression
Zeigt den Ausdruck (Berechnungen, Labels...) hexadezimal und dezimal an.

!Opcode
Hiermit können Sie einzelne Maschienensprachbefehle ausprobieren. (z.B. 1CLR.B (A0)+) Die Wirkung können Sie mit dem Befehl REG überprüfen.

REG
Nach jedem Aufruf von GO, TRACE und !Opcode werden sämtliche Registerinhalte abgespeichert. Mit diesem Befehl werden sie angezeigt.

PMREG
Immer, wenn sich der Rechner wieder einmal "aufgehängt" hat, werden seine letzten Registerinhalte so abgespeichert, daß diese Information nach einem Reset erhalten bleiben. Eben diese Werte zeigt der Befehl an.

QUIT
Hiermit verlassen Sie den Assembler und gelangen zurück zum Desktop

3. Die Assemblerbefehle

Wie schon gesagt, entsprechen die Befehle im großen und ganzen den allgemeinen Regeln. Einige Besonderheiten:

3.1 Der Assembler velangt "klar" formulierte Befehle. Einige Beispiele: Schreiben Sie ADDA A0,Al statt ADD A0,A1; CMPI #5,(A1) statt CMP #5,(A1)usw. MOVEQ verwendet er auch nur dann, wenn es im Listing gewünscht wird.

3.2 Die Wortbreite wird, wie üblich, durch Anhängen von ".B",".W" oder ".L" an den Befehl gekennzeichnet. Wenn nur eine Breite erlaubt ist, oder ".W" gewünscht ist, können Sie darauf auch verzichten.

Bei der Adressierung 'Adressregister indirekt mit Index und Distanz' müssen Sie jedoch die Verarbeitungsbreite angeben. (z.B. .. 0(A0,D0.W))

3.3 Kurze Branch-Befehle mit einer Sprungweite zwischen -128 und +126 sollten Sie mit einem ".S" kennzeichnen, denn nur dann werden Sie auch als kurze Befehle übersetzt.

3.4 Das Kürzel "SP" (Stack pointer) akzeptiert der Assembler nicht. Schreiben Sie statt dessen "A7".

3.5 Zusätzlich gibt es noch die Befehle SAVEALL und LOADALL. Sie sind Abkürzungen für MOVEM.L D0-A6,-(A7) bzw. MOVEM.L (A7)+,D0-A6. Für die LINE A-Befehle stehen ebenfalls Mnemonics zur Verfügung:

```
A_INIT  = $A000,  A_PUPIX = $A001,
A_GEPIX = $A002,  A_LINE  = $A003,
A_HOLIN = $A004,  A_RECT  = $A005,
A_POLY  = $A006,  A_BTBLT = $A007,
A_TXBLT = $A008,  A_SHMOU = $A009,
A_HICUR = $A00A,  A_TRMOU = $A00B,
A_UNSPR = $A00C,  A_DRSPR = $A00D,
A_COPRF = $A00E
```

4. Pseudobefehle

Neben den eigentlichen Assemblerbefehlen können Sie im Listing von weiteren Anweisungen Gebrauch machen:

4.1 Bemerkungen müssen mit einem Semikolon oder einem Sternchen beginnen. Bei einem Abstand von mehreren Zeichen zu dem Assemblerbefehl der selben Zeile, können Sie auch darauf verzichten.

4.2 Der Befehl DC (.DC ist auch erlaubt) definiert eine Reihe von Bytes, Worten oder Langworten (abhängig von .B, .W, .L). Texte müssen dabei mit einfachen oder doppelten Anführungszeichen eingeschlossen werden. Bei .W und .L werden diese ggf. mit einem Nullbyte ergänzt, so daß die nächste Adresse eine geradzahlige ist. Durch Voranstellen eines Ausrufezeichens können Sie in einer Kette von Bytes oder Worten ein Langwort festlegen. (Beispiel: DC.B 1, 2, !LABEL,..)

4.3 Mit DS (oder auch .DS) läßt sich Speicher reservieren. Auch hier können Sie wieder mit .B, .W und .L arbeiten.

4.4 Der Befehl EVEN bewirkt, daß die nächste zu bearbeitende Adresse eine geradzahlige ist.

4.5 Wenn Sie eine ausführbare Programmdatei erstellen wollen (mit PRGASM und PRGSAVE) , müssen Sie drei Sektionen unterscheiden: Eine, in der das Programm steht, eine für intialisierte Daten und eine für unintialisierte Daten. Dieser Assembler verlangt, daß diese Blöcke schon im Listing der genannten Reihenfolge auftauchen. Um sie zu kennzeichnen, schreiben Sie .DATA am Beginn der definierten Daten und .BSS am Beginn der Undefinierten.

5. Labels

Selbstverständlich können Sie auch Labels verwenden. Sie müssen unmittelbar am Zeilenanfang stehen. Im Gegensatz dazu muß vor den Assemblerbefehlen mindestens ein Leerzeichen stehen. Die Labels dürfen beliebig lang sein, allerdings werden nur die ersten 10 Zeichen unterschieden. Die Labels dürfen nicht mit einem Doppelpunkt abgeschlossen werden. Ein Label muß mit einem Buchstaben beginnen und darf weiterhin nur Buchstaben, Zahlen oder das " "- Zeichen enthalten.

6. Editieren

Beim Eingeben eines Direktbefehls oder beim Editieren von Zeilen sollten Sie folgendes beachten:
Die Editierzeile ist stets 72 Zeichen lang. Sollten Sie versuchen, über die Zeile hinauszuschreiben, so wird der Cursor wieder auf den Anfang der Zeile gesetzt.

Mit den Cursortasten können Sie den Cursor nach rechts und nach links bewegen. Mit der Taste "Clr Home" läßt sich die Zeile löschen.
Um Zeichen einzuschieben odpr zu löschen, können Sie von "Insert", "Delete" und "Backspace" Gebrauch machen.

Um die Eingabe zu beenden, drücken Sie "Return". Wollen Sie, daß die geänderte Zeile ignoriert wird, so drücken Sie "Undo". (Bei EDIT, INS und ADD kehren Sie dadurch automatisch in den Direktmodus zurück.) Hier sei auch erwähnt, daß Sie Listvorgänge mit Druck auf die Leertaste anhalten und wieder fortsetzen und mit Return abbrechen können.

7. Zahlensysteme

Überall, wo Sie Werte angeben müssen - sei es im Assemblerlisting oder im Direktmodus - haben Sie verschiedene Möglichkeiten der Darstellung zur Auswahl:

Dezimal	: (+/-)xxxx
Hexadezimal	: (+/-)$xxxx
Binär	: (+/-)%xxxx
Label	: LABEL
Character	: 'c
String (bis zu 4 Zeichen)	: "cccc"

Außerdem können Sie mehrere solcher Werte (außer Character- und Stringwerten) mit den Rechenoperationen +, -, *, / verknüpfen. Es gelten dabei nicht

die üblichen Rechenregeln, sondern die Ergebnisse werden von rechts nach links berechnet! (Beispiel: Schreiben Sie anstatt "(b-c)*a:" "a*b-c").
Dies ist sicherlich eine sehr ungewohnte und unübersichtliche Schreibweise. Da ich jedoch davon ausging, daß man in der Regel höchstens von einer einfachen Addition oder Subtraktion Gebrauch macht, habe ich mir die Mühe gespart, eine bessere Berechnung zu programmieren.

8. Fehlermeldungen

Während des Assemblierens oder auch während der Direkteingabe kann es zu verschiedenen Fehlern kommen. Es ertönt dann ein Glockenton und eine Fehlermeldung erscheint. Das Assemblieren wird dabei abgebrochen.
Es kann nicht garantiert werden, daß der Assembler absolut alle Fehler erkennt, aber in weit den meisten Fällen dürften solche Probleme nicht auftauchen. (Wenn Sie sich im Unklaren sind, können Sie die fragliche Stelle ja einmal disassemblieren.)
Hier nun die Fehlermeldungen mit ihren Bedeutungen:

```
Illegal opcode       : Beim Assemblieren tauchte ein
                       unbekannter Befehl auf.
Undefined error      : Nicht näher identifizierbarer
                       Fehler.
Wrong adressing mode : Der gewünschte
                       Adressierungsmodus darf an
                       dieser Stelle nicht angewandt
                       werden.
Unknown adr. mode    : Der Adressierungemodus
                       existiert nicht.
Undefined label      : Es wird auf ein nicht
                       definiertes Label verwiesen.
Syntax                 : Fehlerhafte Eingabe im
                       Direktmodus.
File not found       : Datei existiert nicht.
```

Meldung	Bedeutung
Double def. label	: Label wurde zum zweiten Mal definiert.
Illegal reg #	: Sie haben in Verbindung mit einem Adress- oder Daten-register eine falsche Nummer angegeben. (0-7 sind zulässig.)
Adr. out of ränge	: Ein Branch-Befehl verzweigt zu einer Adresse, die nicht in seinem Bereich liegt. Dieser Fehler kann auch auftauchen, wenn Sie mit einem Branch.S Befehl unmittelbar zur nächsten Adresse springen wollen.
Value out of range	: Der Wert kann mit der gewählten Wortbreite nicht dargestellt werden.
Abort	: Sie haben den Assemblier-vorgang mit einem Tastendruck abgebrochen.
Illegal word size	: Die gewünschte Wortbreite ist bei dem Befehl nicht erlaubt.
Printer not on line	: Der Drucker ist nicht eingeschaltet oder aus anderen Gründen nicht empfangsbereit.
Unknown word size	: Sie haben eine andere Verarbeitungsbreite als .B, .W oder .L angegeben.
No data register	: Sie müssen bei dem Befehl ein Datenregister verwenden.
Disk: -xx	: Es tauchte der mit der Nummer angegebene Diskettenfehler auf.

C. Mitgelieferte Programmdiskette

Atari ST ASCII-Tabelle															
	0		32	@	64	`	96	Ç	128	á	160	ĳ	192	α	224
⇧	1	!	33	A	65	a	97	ü	129	í	161	Ĳ	193	β	225
⇩	2	"	34	B	66	b	98	é	130	ó	162	א	194	Γ	226
⇨	3	#	35	C	67	c	99	â	131	ú	163	ב	195	π	227
⇦	4	$	36	D	68	d	100	ä	132	ñ	164	ג	196	Σ	228
☒	5	%	37	E	69	e	101	à	133	Ñ	165	ד	197	σ	229
◩	6	&	38	F	70	f	102	å	134	ª	166	ה	198	µ	230
⌛	7	'	39	G	71	g	103	ç	135	º	167	ו	199	τ	231
✓	8	(	40	H	72	h	104	ê	136	¿	168	ז	200	Φ	232
◷	9	)	41	I	73	i	105	ë	137	⌐	169	ח	201	Θ	233
🔔	10	*	42	J	74	j	106	è	138	¬	170	ט	202	Ω	234
♪	11	+	43	K	75	k	107	ï	139	½	171	י	203	δ	235
␌	12	,	44	L	76	l	108	î	140	¼	172	כ	204	∮	236
␍	13	-	45	M	77	m	109	ì	141	¡	173	ל	205	ϕ	237
◢	14	.	46	N	78	n	110	Ä	142	«	174	מ	206	∈	238
◣	15	/	47	O	79	o	111	Å	143	»	175	נ	207	∩	239
0	16	0	48	P	80	p	112	É	144	ã	176	ס	208	≡	240
1	17	1	49	Q	81	q	113	æ	145	õ	177	ע	209	±	241
2	18	2	50	R	82	r	114	Æ	146	Ø	178	פ	210	≥	242
3	19	3	51	S	83	s	115	ô	147	ø	179	צ	211	≤	243
4	20	4	52	T	84	t	116	ö	148	œ	180	ק	212	⌠	244
5	21	5	53	U	85	u	117	ò	149	Œ	181	ר	213	⌡	245
6	22	6	54	V	86	v	118	û	150	À	182	ש	214	÷	246
7	23	7	55	W	87	w	119	ù	151	Ã	183	ת	215	≈	247
8	24	8	56	X	88	x	120	ÿ	152	Õ	184	ן	216	°	248
9	25	9	57	Y	89	y	121	Ö	153	¨	185	ך	217	•	249
ə	26	:	58	Z	90	z	122	Ü	154	´	186	ם	218	·	250
␛	27	;	59	[	91	{	123	¢	155	†	187	ף	219	√	251
	28	<	60	\	92	|	124	£	156	¶	188	ץ	220	ⁿ	252
	29	=	61	]	93	}	125	¥	157	©	189	§	221	²	253
	30	>	62	^	94	~	126	ß	158	®	190	∧	222	³	254
	31	?	63	_	95	△	127	ƒ	159	™	191	∞	223	¯	255

D. ASCII-Tabelle

Anhang E
Befehlskode in numerischer Reihenfolge

Hier folgt nochmal eine Befehlsübersicht, aber diesmal geordnet in numerischer Reihenfolge. Mit Hilfe dieser Tabelle können Sie an Ihrem Schreibtisch aus Objectkode Befehle disassemblieren.

Die Tabelle enthält nach Anhang B eigentlich keine neue Information: wir haben sie nur zusammengefaßt und die Reihenfolge verändert.

Aus der Tabelle wird auch ersichtlich, WARUM manche Adressierungsmoden "nicht erlaubt" sind: Der entsprechende Kode ist anders zugeordnet.

Die Doppelbelegung der Tabelle an manchen Stellen ist daher nur scheinbar.

Alle Kode, die nicht in dieser Tabelle enthalten sind, sollten eine Illegal Exception auslösen.

15		13		11		9	8	7	6	5	4	3	2	1	0	Bit
0	0	0	0	0	0	0	0	x	x	x	x	x	x	x	x	ORI
0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	0	ORI zum CCR
0	0	0	0	0	0	0	0	0	1	1	1	1	1	0	0	ORI zum SR
0	0	0	0	x	x	x	1	0	0	x	x	x	x	x	x	BTST
0	0	0	0	x	x	x	1	0	1	x	x	x	x	x	x	BCHG
0	0	0	0	x	x	x	1	1	0	x	x	x	x	x	x	BCLR
0	0	0	0	x	x	x	1	1	1	x	x	x	x	x	x	BSET
0	0	0	0	x	x	x	1	x	x	0	0	1	x	x	x	MOVEP
0	0	0	0	0	0	1	0	x	x	x	x	x	x	x	x	ANDI
0	0	0	0	0	0	1	0	0	0	1	1	1	1	0	0	ANDI zum CCR
0	0	0	0	0	0	1	0	0	1	1	1	1	1	0	0	ANDl zum SR
0	0	0	0	0	1	0	0	x	x	x	x	x	x	x	x	SUBI
0	0	0	0	0	1	1	0	x	x	x	x	x	x	x	x	ADDI
0	0	0	0	1	0	0	0	0	0	x	x	x	x	x	x	BTST
0	0	0	0	1	0	0	0	0	1	x	x	x	x	x	x	BCHG
0	0	0	0	1	0	0	0	1	0	x	x	x	x	x	x	BCLR
0	0	0	0	1	0	0	0	1	1	x	x	x	x	x	x	BSET
0	0	0	0	1	0	1	0	x	x	x	x	x	x	x	x	EORI
0	0	0	0	1	0	1	0	0	0	1	1	1	1	0	0	EORI zum CCR
0	0	0	0	1	0	1	0	0	1	1	1	1	1	0	0	EORI zum SR
0	0	0	0	1	1	0	0	x	x	x	x	x	x	x	x	CMPI
0	0	0	1	x	x	x	x	x	x	x	x	x	x	x	x	MOVE
0	0	1	x	x	x	x	x	x	x	x	x	x	x	x	x	MOVE
0	1	0	0	0	0	0	0	x	x	x	x	x	x	x	x	NEGX
0	1	0	0	0	0	0	0	1	1	x	x	x	x	x	x	MOVE VOM SR
0	1	0	0	x	x	x	1	1	0	x	x	x	x	x	x	CHK
0	1	0	0	x	x	x	1	1	1	x	x	x	x	x	x	LEA
0	1	0	0	0	0	1	0	x	x	x	x	x	x	x	x	CLR
0	1	0	0	0	1	0	0	x	x	x	x	x	x	x	x	NEG
0	1	0	0	0	1	0	0	1	1	x	x	x	x	x	x	MOVE zum CCR
0	1	0	0	0	1	1	0	x	x	x	x	x	x	x	x	NOT
0	1	0	0	0	1	1	0	1	1	x	x	x	x	x	x	MOVE zum SR
0	1	0	0	1	0	0	0	0	0	x	x	x	x	x	x	NBCD
0	1	0	0	1	0	0	0	0	1	0	0	0	x	x	x	SWAP
0	1	0	0	1	0	0	0	0	1	x	x	x	x	x	x	PEA
0	1	0	0	1	0	0	0	1	x	0	0	0	x	x	x	EXT

15		13		11		9	8	7	6	5	4	3	2	1	0	Bit
0	1	0	0	1	0	0	0	1	x	x	x	x	x	x	x	MOVEM
0	1	0	0	1	0	1	0	x	x	x	x	x	x	x	x	TST
0	1	0	0	1	0	1	0	1	1	x	x	x	x	x	x	TAS
0	1	0	0	1	0	1	0	1	1	1	1	1	1	0	0	ILLEGAL
0	1	0	0	1	1	0	0	1	x	x	x	x	x	x	x	MOVEM
0	1	0	0	1	1	1	0	0	1	0	0	x	x	x	x	TRAP
0	1	0	0	1	1	1	0	0	1	0	1	0	x	x	x	LINK
0	1	0	0	1	1	1	0	0	1	0	1	1	x	x	x	UNLK
0	1	0	0	1	1	1	0	0	1	1	0	0	x	x	x	MOVE zum USP
0	1	0	0	1	1	1	0	0	1	1	0	1	x	x	x	MOVE vom USP
0	1	0	0	1	1	1	0	0	1	1	1	0	0	0	0	RESET
0	1	0	0	1	1	1	0	0	1	1	1	0	0	0	1	NOP
0	1	0	0	1	1	1	0	0	1	1	1	0	0	1	0	STOP
0	1	0	0	1	1	1	0	0	1	1	1	0	0	1	1	RTE
0	1	0	0	1	1	1	0	0	1	1	1	0	1	0	1	RTS
0	1	0	0	1	1	1	0	0	1	1	1	0	1	1	0	TRAPV
0	1	0	0	1	1	1	0	0	1	1	1	0	1	1	1	RTR
0	1	0	0	1	1	1	0	1	0	x	x	x	x	x	x	JSR
0	1	0	0	1	1	1	0	1	1	x	x	x	x	x	x	JMP
0	1	0	1	x	x	x	0	x	x	x	x	x	x	x	x	ADDQ
0	1	0	1	x	x	x	x	1	1	x	x	x	x	x	x	Scc
0	1	0	1	x	x	x	x	1	1	0	0	1	x	x	x	DBcc
0	1	0	1	x	x	x	1	x	x	x	x	x	x	x	x	SUBQ
0	1	1	0	x	x	x	x	x	x	x	x	x	x	x	x	Bcc
0	1	1	0	0	0	0	1	x	x	x	x	x	x	x	x	BSR
0	1	1	1	x	x	x	0	x	x	x	x	x	x	x	x	MOVEQ
1	0	0	0	x	x	x	x	x	x	x	x	x	x	x	x	OR
1	0	0	0	x	x	x	0	1	1	x	x	x	x	x	x	DIVU
1	0	0	0	x	x	x	1	0	0	0	0	x	x	x	x	SBCD
1	0	0	0	x	x	x	1	1	1	x	x	x	x	x	x	DIVS
1	0	0	1	x	x	x	x	x	x	x	x	x	x	x	x	SUB
1	0	0	1	x	x	x	x	1	1	x	x	x	x	x	x	SUBA
1	0	0	1	x	x	x	1	x	x	0	0	x	x	x	x	SUBX
1	0	1	0	x	x	x	x	x	x	x	x	x	x	x	x	Exception 10
1	0	1	1	x	x	x	0	x	x	x	x	x	x	x	x	CMP
1	0	1	1	x	x	x	x	1	1	x	x	x	x	x	x	CMPA

E. Befehlscode in numerischer Reihenfolge

15		13		11		9	8	7	6	5	4	3	2	1	0	Bit
1	0	1	i	x	x	x	1	x	x	x	x	x	x	x	x	EOR
1	0	1	i	x	x	x	1	x	x	0	0	1	x	x	x	CMPM
1	1	0	0	x	x	x	x	x	x	x	x	x	x	x	x	AND
1	1	0	0	x	x	x	0	1	1	x	x	x	x	x	x	MULU
1	1	0	0	x	x	x	1	0	0	0	0	x	x	x	x	ÄBCD
1	1	0	0	x	x	x	1	0	1	0	0	0	x	x	x	EXG
1	1	0	0	x	x	x	1	1	0	0	0	1	x	x	x	EXG
1	1	0	0	x	x	x	1	1	1	x	x	x	x	x	x	MULS
1	1	0	1	x	x	x	x	x	x	x	x	x	x	x	x	ADD
1	1	0	1	x	x	x	x	1	1	x	x	x	x	x	x	ADDA
1	1	0	1	x	x	x	1	x	x	0	0	x	x	x	x	ADDX
1	1	1	0	0	0	0	0	1	1	x	x	x	x	x	x	ASR
1	1	1	0	0	0	0	1	1	1	x	x	x	x	x	x	ASL
1	1	1	0	0	0	1	0	1	1	x	x	x	x	x	x	LSR
1	1	1	0	0	0	1	1	1	1	x	x	x	x	x	x	LSL
1	1	1	0	0	1	0	0	1	1	x	x	x	x	x	x	ROXR
1	1	1	0	0	1	0	1	1	1	x	x	x	x	x	x	ROXL
1	1	1	0	0	1	1	0	1	1	x	x	x	x	x	x	ROR
1	1	1	0	0	1	1	1	1	1	x	x	x	x	x	x	ROL
1	1	1	0	x	x	x	0	x	x	x	0	0	x	x	x	ASR
1	1	1	0	x	x	x	1	x	x	x	0	0	x	x	x	ASL
1	1	1	0	x	x	x	0	x	x	x	0	1	x	x	x	LSR
1	1	1	0	x	x	x	1	x	x	x	0	1	x	x	x	LSL
1	1	1	0	x	x	x	0	x	x	x	1	0	x	x	x	ROXR
1	1	1	0	x	x	x	1	x	x	x	1	0	x	x	x	ROXL
1	1	1	0	x	x	x	0	x	x	x	1	1	x	x	x	ROR
1	1	1	0	x	x	x	1	x	x	x	1	1	x	x	x	ROL
1	1	1	1	x	x	x	x	x	x	x	x	x	x	x	x	Exception 11

Anhang F

Stichwortverzeichnis

F. Stichwortverzeichnis

Über die 2. Auflage dieses Buches



Seit der Erstauflage von 1989 hat sich in der Computerwelt vieles verändert. Waren damals die 16-Bit Prozessoren ziemlich neu, so sind es jetzt (2010) die 64-Bit Prozessoren. Die Notwendigkeit in Assembler zu programmieren ist inzwischen kleiner geworden, einerseits, weil die Prozessoren so viel schneller geworden sind daß die Rechenzeit keine Rolle mehr spielt, andererseits, weil die Unterstützung etwa durch den neuesten C-Compiler so viel besser geworden ist, daß man damit schneller und zuverlässiger zum Ergebnis kommt.

Und den Atari ST und den tollen Prozessor in diesem Buch gibt's nicht mehr, allerdings hätte es ohne die damaligen Prozessoren die heutigen Prozessoren nicht gegeben, denn man braucht bekanntlich einen Computer, um einen Computer entwerfen zu können.

Und den Heim Verlag gibt's nicht mehr. Und den Nachfolgeverlag im Ortsmitte von Darmstadt-Eberstadt gibt's auch nicht mehr. Ich habe mich daher entschieden, dieses entsprechend obsolet gewordenes Buch selber aufzulegen, damit ich es mir nochmal anschauen und anderen zeigen kann.

Joris Teepe

Kleve, 2010

68000 ASSEMBLER

Einführung in die ASSEMBLER-PROGRAMMIERUNG

WICHTIGE MERKMALE:

- Dieses Buch ist eine Einführung und damit für alle ATARI-Besitzer ein leicht verständlicher Einstieg in die Möglichkeiten der ASSEMBLER-PROGRAMMIERUNG. Vorausgesetzt wird die Beherrschung wenigstens einer höheren Computersprache, die Bedienung des Betriebssystems und das Rechnen mit binären und hexadezimalen Zahlen.
- Großer Wert wurde auf eine klare Sprache und Darstellung gelegt. So wurde Fachjargon vermieden. Alle neuen Fachbegriffe werden deutlich und didaktisch definiert.
- Bewußt ist das Eingangsniveau niedrig gehalten. Kenntnisse von anderen ASSEMBLERN sind nicht erforderlich. Dies erleichtert hervorragend den Einstieg in die Maschinensprache.
- Bei der Beschreibung der Befehle wird in der Erklärung auf Befehle mit ähnlicher Wirkung hingewiesen. Weiterhin erhält der Leser hier umfangreiche Informationen und bei auftauchenden Begriffen einen Verweis auf das entsprechend Kapitel im Buch.
- Insg esamt sind im Buch **mehrere hundert Querverweise** aufgenommen. Dadurch ist man an jeder Stelle dieses Buches imstande, sich die nötige Information zu einem Begriff oder einer Erklärung nachzuschlagen.
- Das Buch will für seine Leser auch eine Einführung zur Nutzung und Anwendung weiterer Literatur sein. **Deshalb zeichnet es sich auch ganz besonders dadurch aus, daß zu den definierten deutschen Fachbegriffen auch der entsprechende englische Ausdruck gezeigt wird.**
- Wo es erforderlich war, wird im Buch auch auf die Hardware des Computers Bezug genommen.
- **Damit der Leser das Erlernte auch gleich in die Praxis umsetzen kann, enthält das Buch eine Diskette mit einer INTERAKTIVEN ASSEMBLER-ENTWICKLUNGSSOFTWARE.**
- Ein Buch für alle ATARI-Anwender, die den richtigen Einstieg in die ASSEMBLER-PROGRAMMIERUNG suchen.

ISBN 3-923250-77-0
Bestell-Nr. B-436
DM 59,–

Auf 3½"-Diskette enthalten:
INTERAKTIVES
ASSMELBER
ENTWICKLUNGSSYSTEM

